

US010169267B2

(12) **United States Patent**
Bradbury et al.

(10) **Patent No.:** **US 10,169,267 B2**
(45) **Date of Patent:** **Jan. 1, 2019**

(54) **TRANSACTIONAL EXECUTION ENABLED SUPERVISOR CALL INTERRUPTION WHILE IN TX MODE**

(56) **References Cited**

U.S. PATENT DOCUMENTS

(71) Applicant: **International Business Machines Corporation**, Armonk, NY (US)
(72) Inventors: **Jonathan D. Bradbury**, Poughkeepsie, NY (US); **Dan F. Greiner**, San Jose, CA (US); **Michael Karl Gschwind**, Chappaqua, NY (US); **Chung-Lung K. Shum**, Wappingers Falls, NY (US)

5,715,458 A * 2/1998 Holder G06F 9/45537
719/319
6,349,361 B1 * 2/2002 Altman G06F 9/3834
711/121
6,381,606 B1 * 4/2002 Carpenter G06F 17/3056
8,316,194 B2 * 11/2012 Adl-Tabatabai G06F 9/466
711/141
9,170,844 B2 10/2015 Blainey et al.
(Continued)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

OTHER PUBLICATIONS

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 396 days.

IBM: List of IBM Patents or Patent Applications Treated as Related (Appendix P), Sep. 4, 2015, pp. 1-2.
(Continued)

(21) Appl. No.: **14/845,350**

Primary Examiner — William B Partridge

(22) Filed: **Sep. 4, 2015**

(74) *Attorney, Agent, or Firm* — Michael A. Petrocelli; Steven Chiu

(65) **Prior Publication Data**

US 2015/0378942 A1 Dec. 31, 2015

(57) **ABSTRACT**

Related U.S. Application Data

(63) Continuation of application No. 14/316,894, filed on Jun. 27, 2014, now Pat. No. 9,524,257.

A computer can manage an interruption while a processor is executing a transaction in a transactional-execution (TX) mode. Execution, in a program context, of the transaction is begun by a processor in TX mode. An interruption request is detected for an interruption, by the processor, in TX mode. The interruption is accepted by the processor to execute a TX compatible routine in a supervisor context for changing supervisor resources. The TX compatible routine is executed within the TX mode. The processor returns to the program context to complete the execution of the transaction. Based on the transaction aborting, the processor does not commit changes to the supervisor resources.

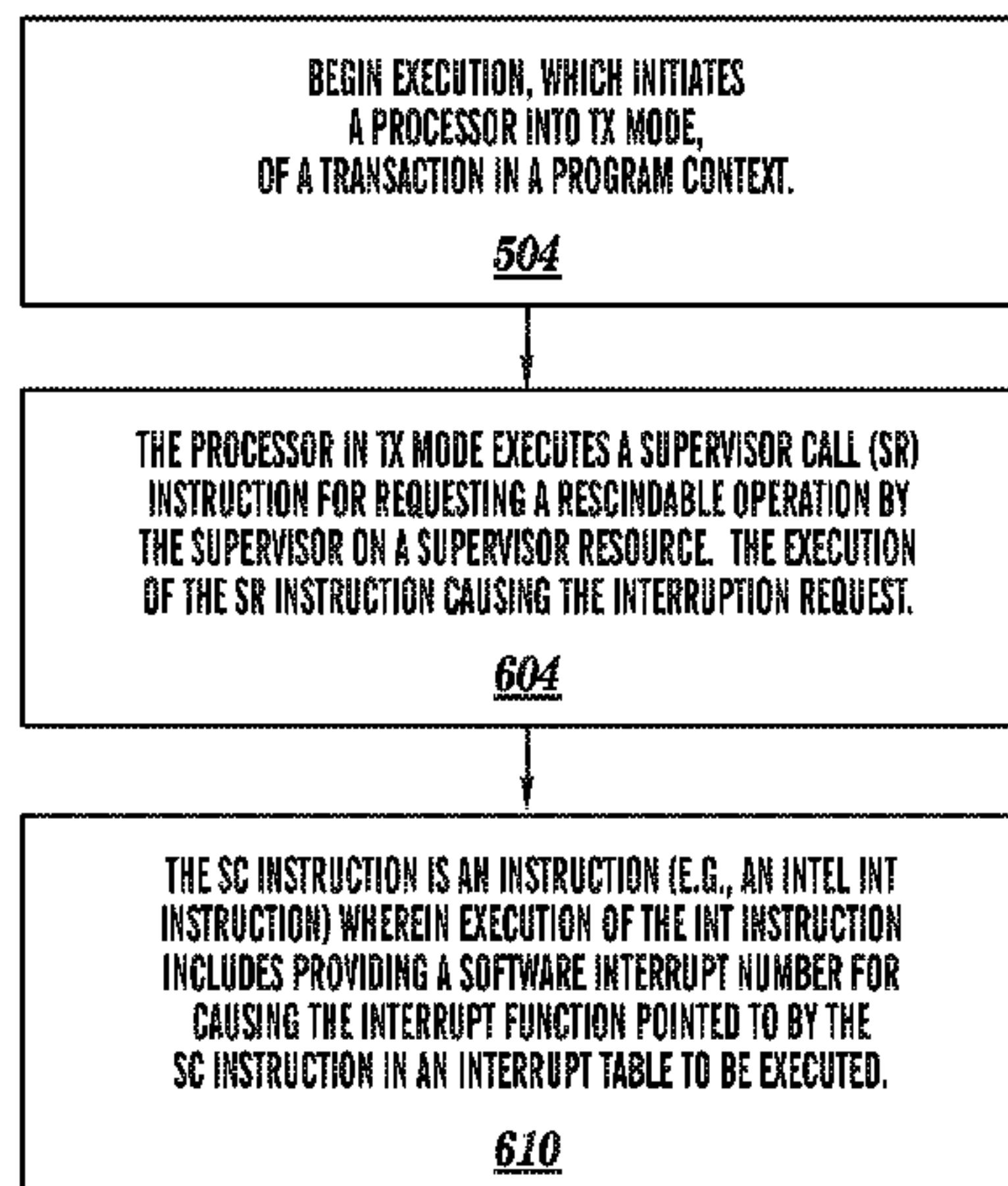
(51) **Int. Cl.**
G06F 9/30 (2006.01)
G06F 13/24 (2006.01)
G06F 9/46 (2006.01)

(52) **U.S. Cl.**
CPC **G06F 13/24** (2013.01); **G06F 9/30145** (2013.01); **G06F 9/466** (2013.01)

(58) **Field of Classification Search**
None
See application file for complete search history.

6 Claims, 8 Drawing Sheets

600 ↙



(56)

References Cited

U.S. PATENT DOCUMENTS

2010/0169528 A1 7/2010 Kumar et al.
 2012/0159028 A1 6/2012 Zimmer et al.
 2013/0247010 A1* 9/2013 Bradbury G06F 9/3005
 717/130
 2013/0339327 A1* 12/2013 Belmar G06F 17/30371
 707/703
 2013/0339687 A1* 12/2013 Greiner G06F 9/3857
 712/229

OTHER PUBLICATIONS

U.S. Appl. No. 14/316,894, filed Jun. 27, 2014, entitled: "Transactional Execution Enabled Supervisor Call Interruption While in TX Mode", pp. 1-95.
 Proceedings 2012 IEEE/ACM 45th International Symposium on Microarchitecture MICRO-45, presented Dec. 1-5, 2012, "Transactional Memory Architecture and Implementation for IBM System z", pp. 25-36.
 IBM, "Principles of Operation", Tenth Edition (Sep. 2012), SA22-7832-09.

"Intel® Architecture Instruction Set Extensions Programming Reference", 319433-012A, Feb. 2012.

McDonald, "Architectures for Transactional Memory", A Dissertation Submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy, Jun. 2009, pp. 1-145.

Mak et al., IBM J. Res. & Dev. vol. 53, No. 1, Paper 2, 2009, "IBM System z10 Processor Cache Subsystem Microarchitecture", pp. 2:1-2:12.

Zick, "User Extended Supervisor Call Router", An IP.com Prior Art Database Technical Disclosure, IP.com No. IPCOM000084610D, IP Electronic Publication: Mar. 2, 2005, pp. 1-6.

Herlihy et al., "Transactional Memory: Architectural Support for Lock-Free Data Structures", ISCA '93 Proceedings of the 20th Annual International Symposium on Computer Architecture, 1993, pp. 1-12.

Pending U.S. Appl. No. 14/316,894, filed Jun. 27, 2014, entitled: "Transactional Execution Enabled Supervisor Call Interruption While in TX Mode", pp. 1-95.

* cited by examiner

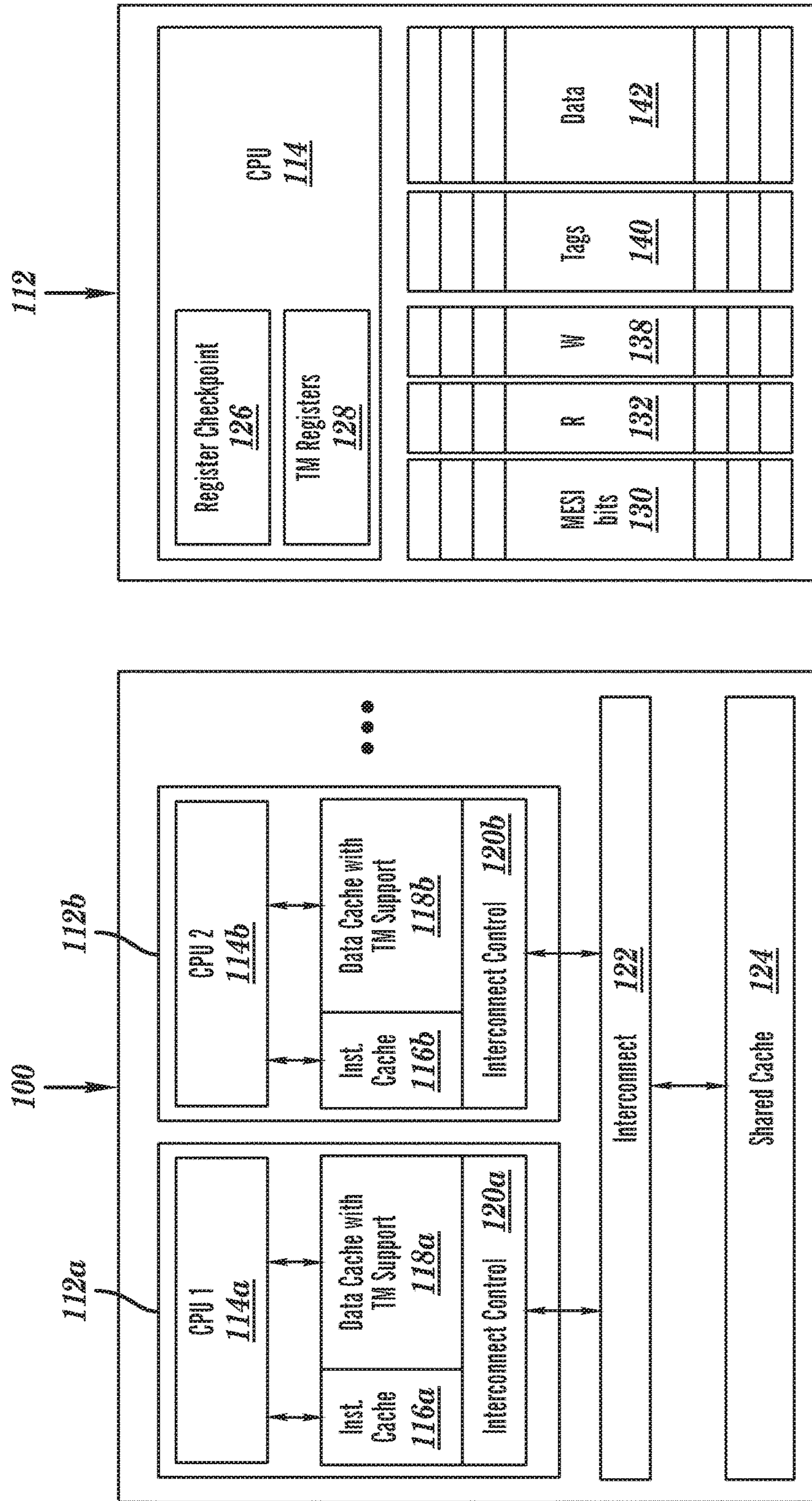
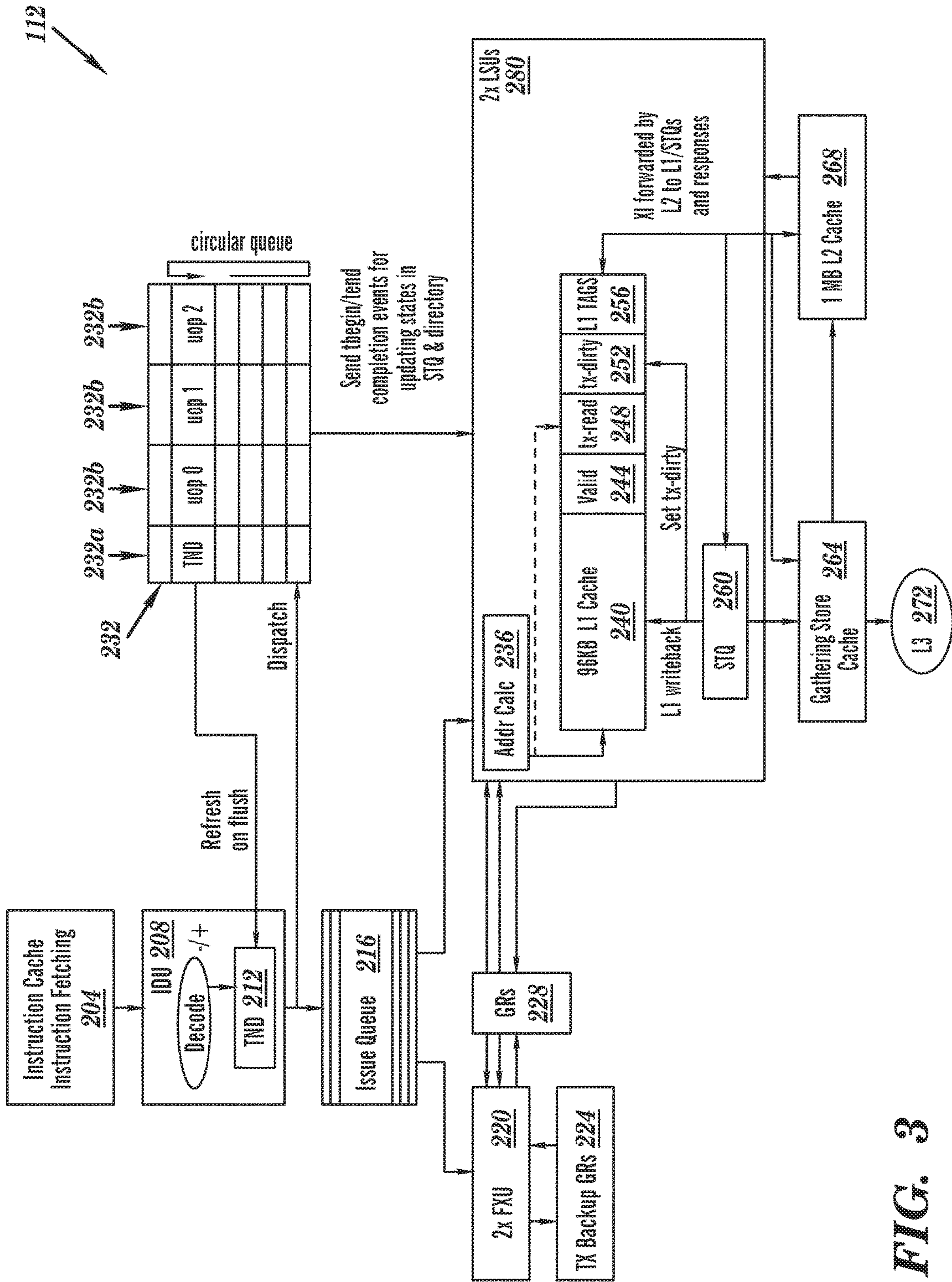


FIG. 1

FIG. 2



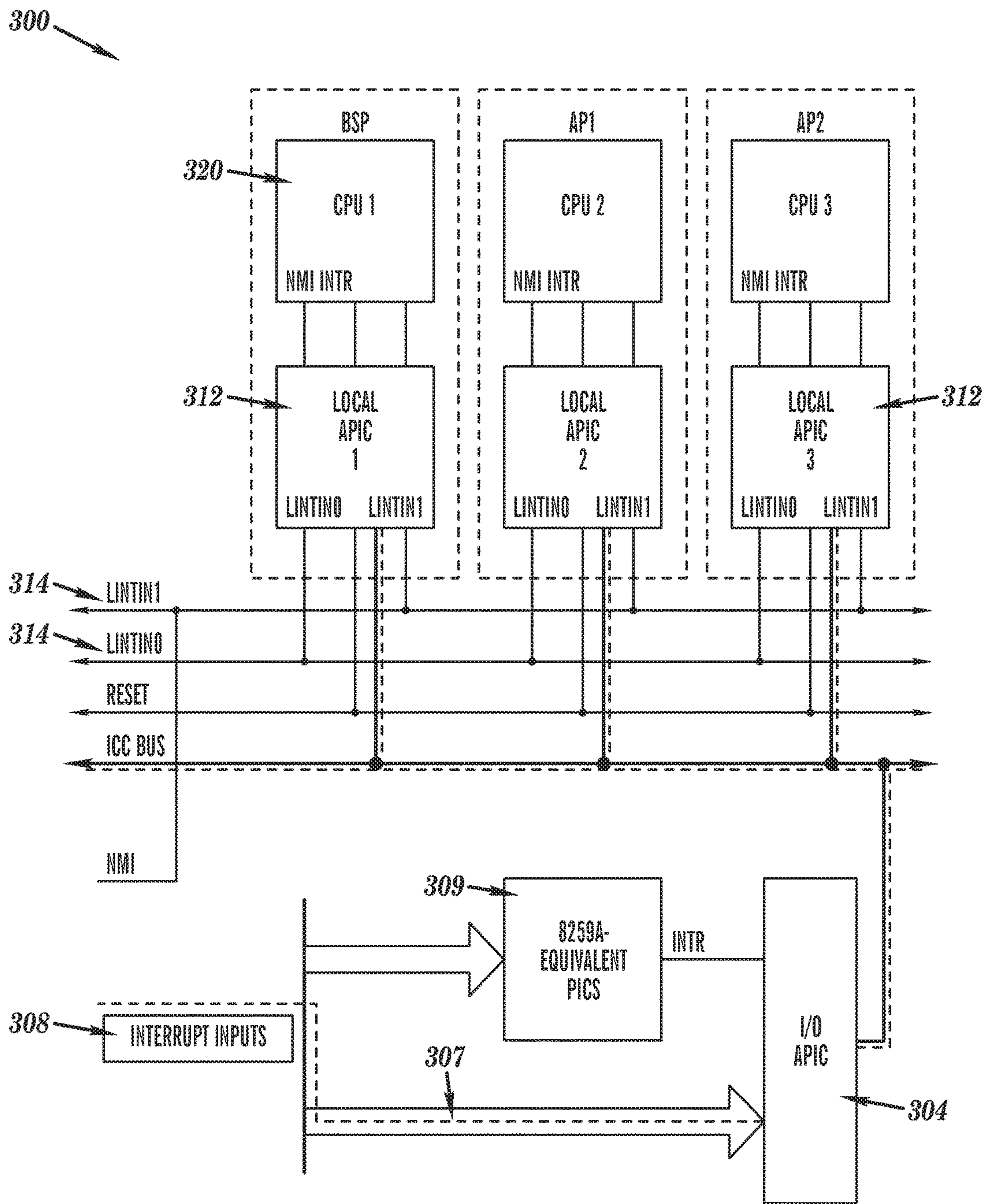


FIG. 4

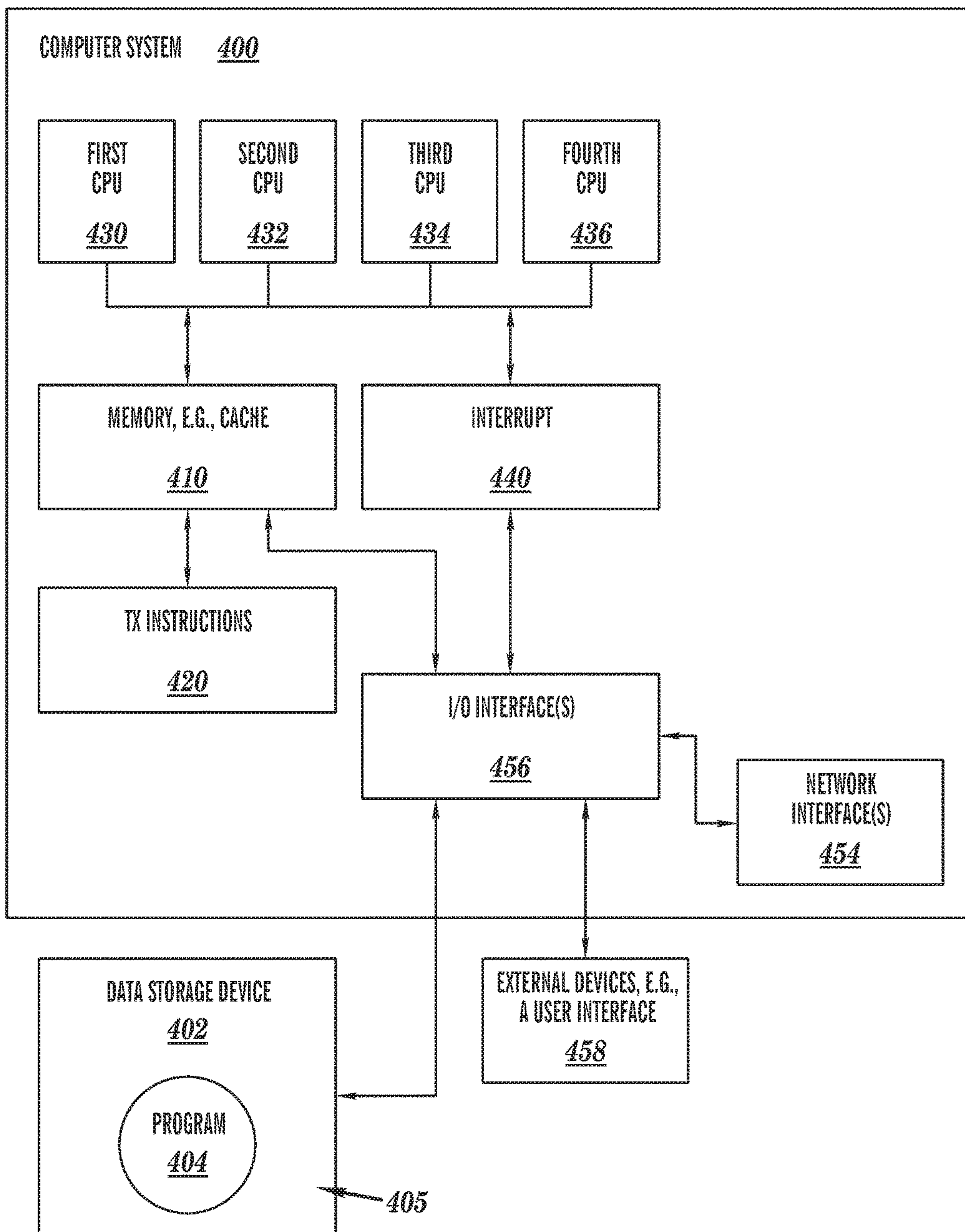


FIG. 5

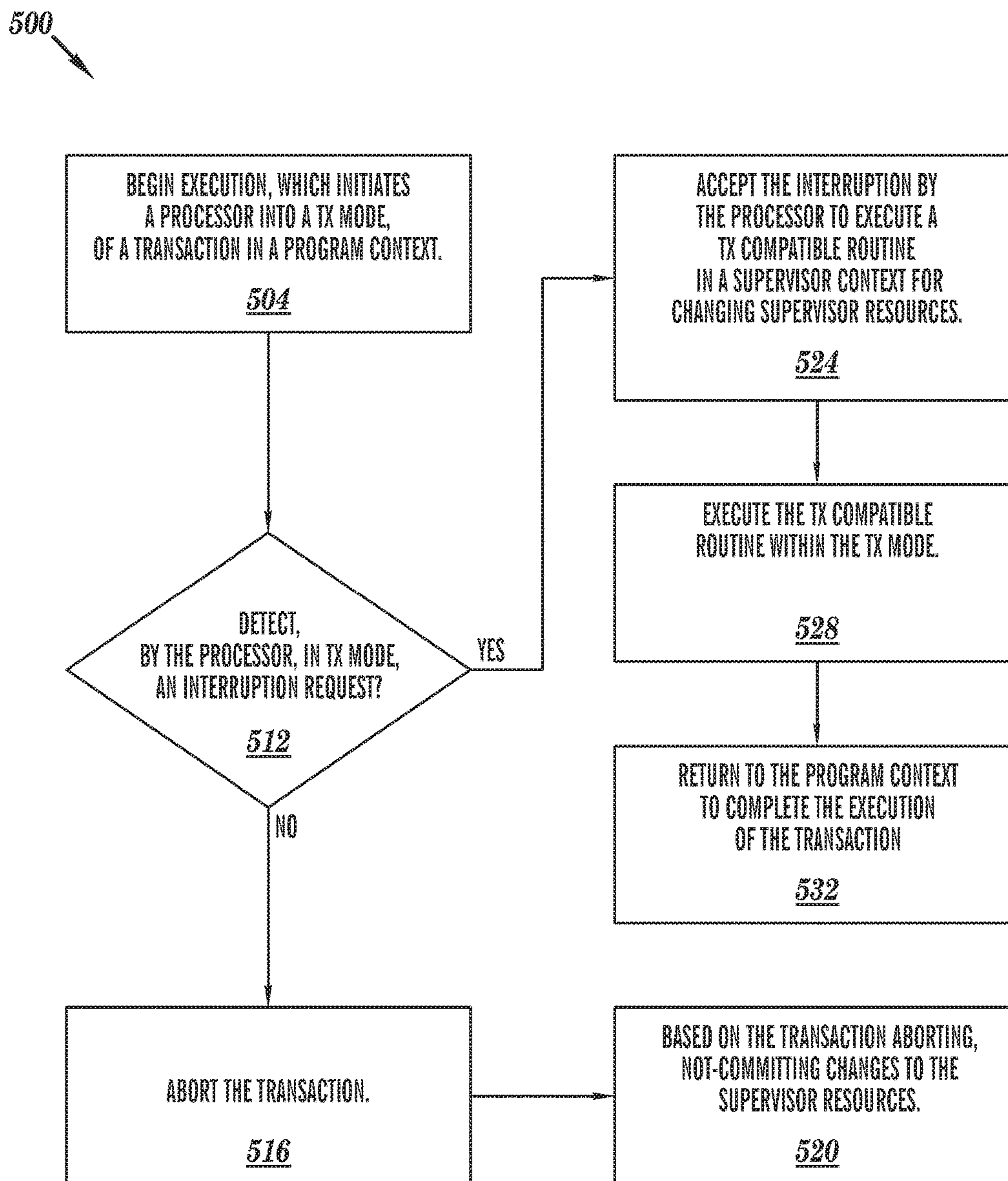
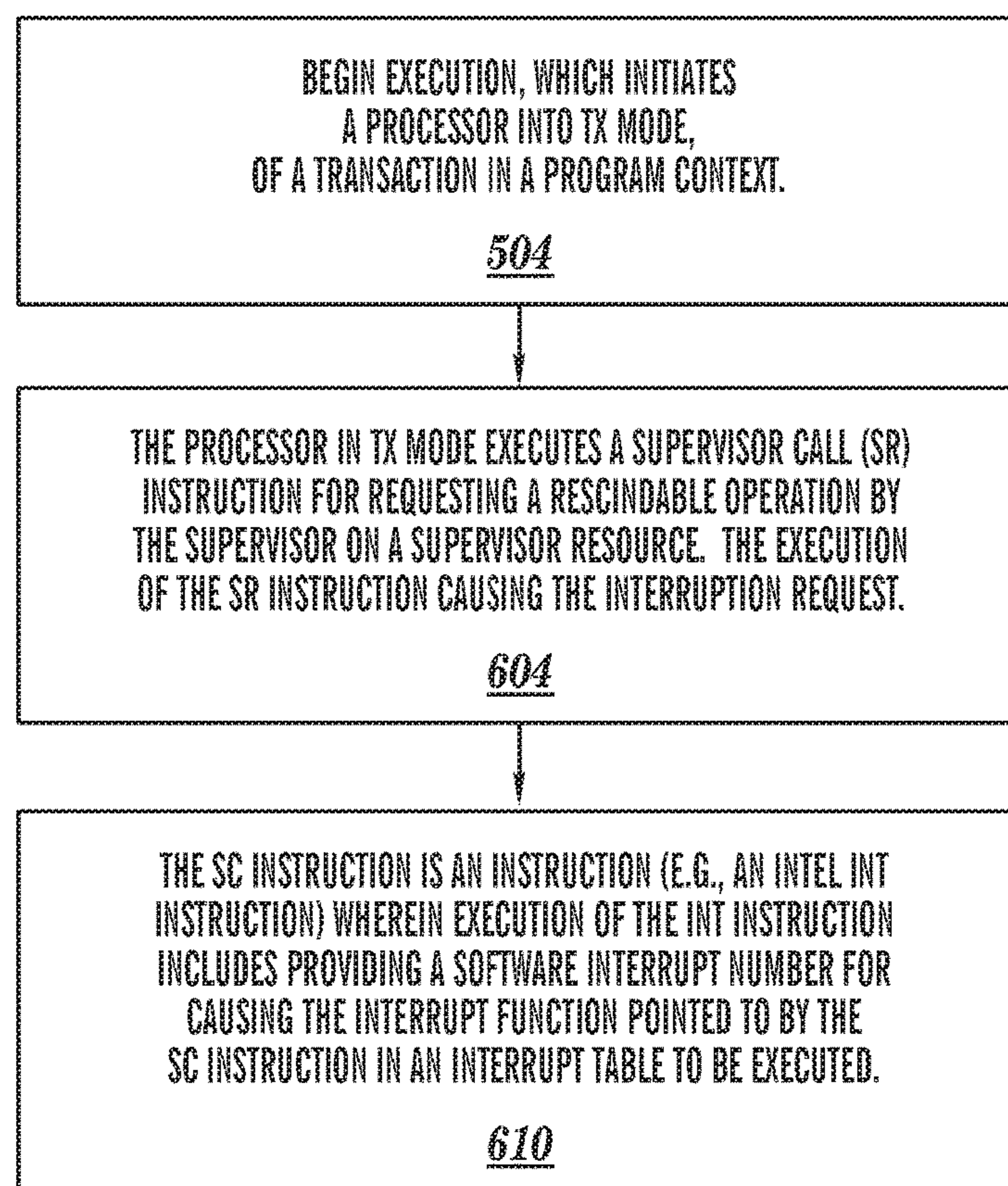


FIG. 6

600

**FIG. 7**

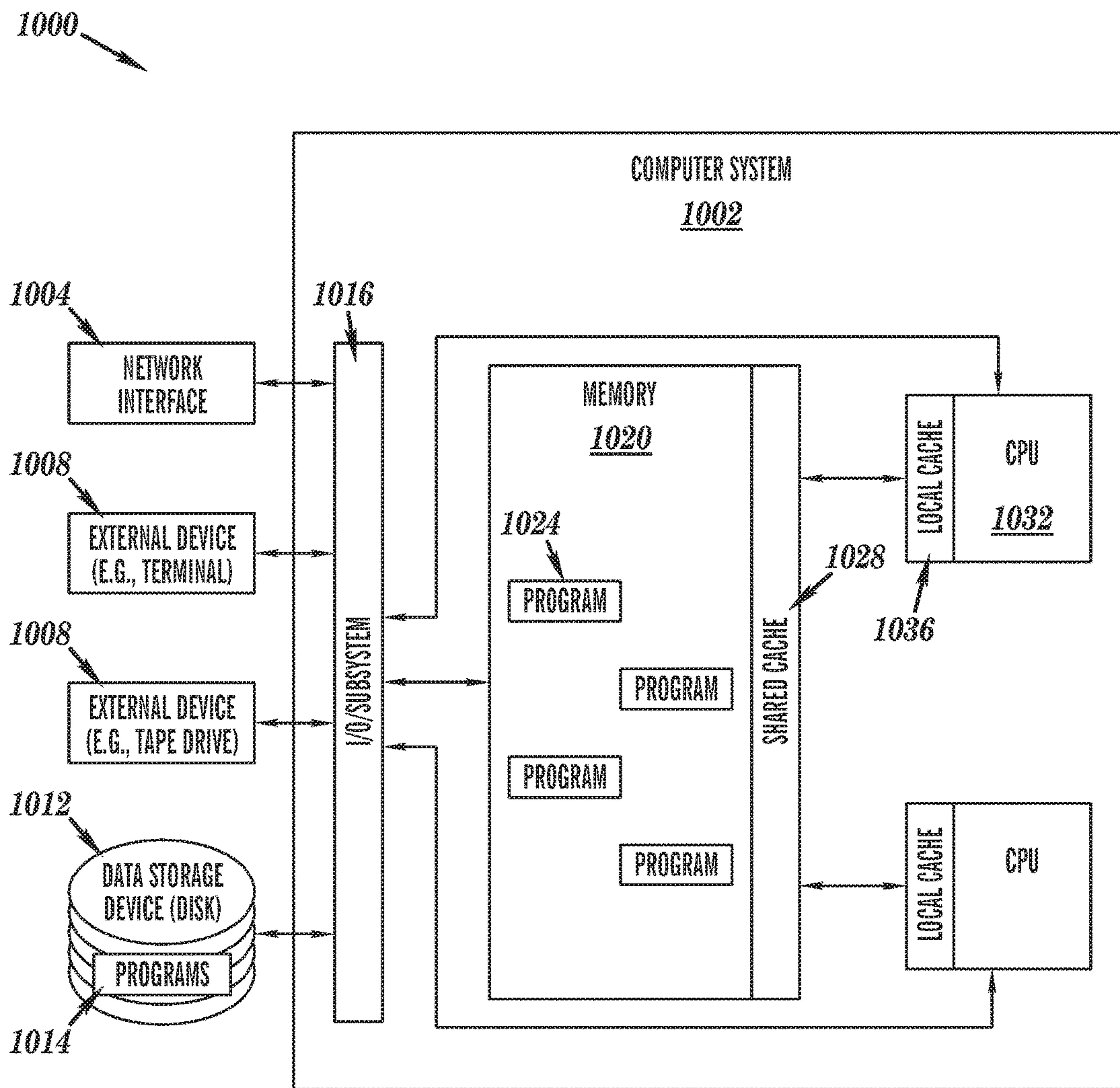


FIG. 8

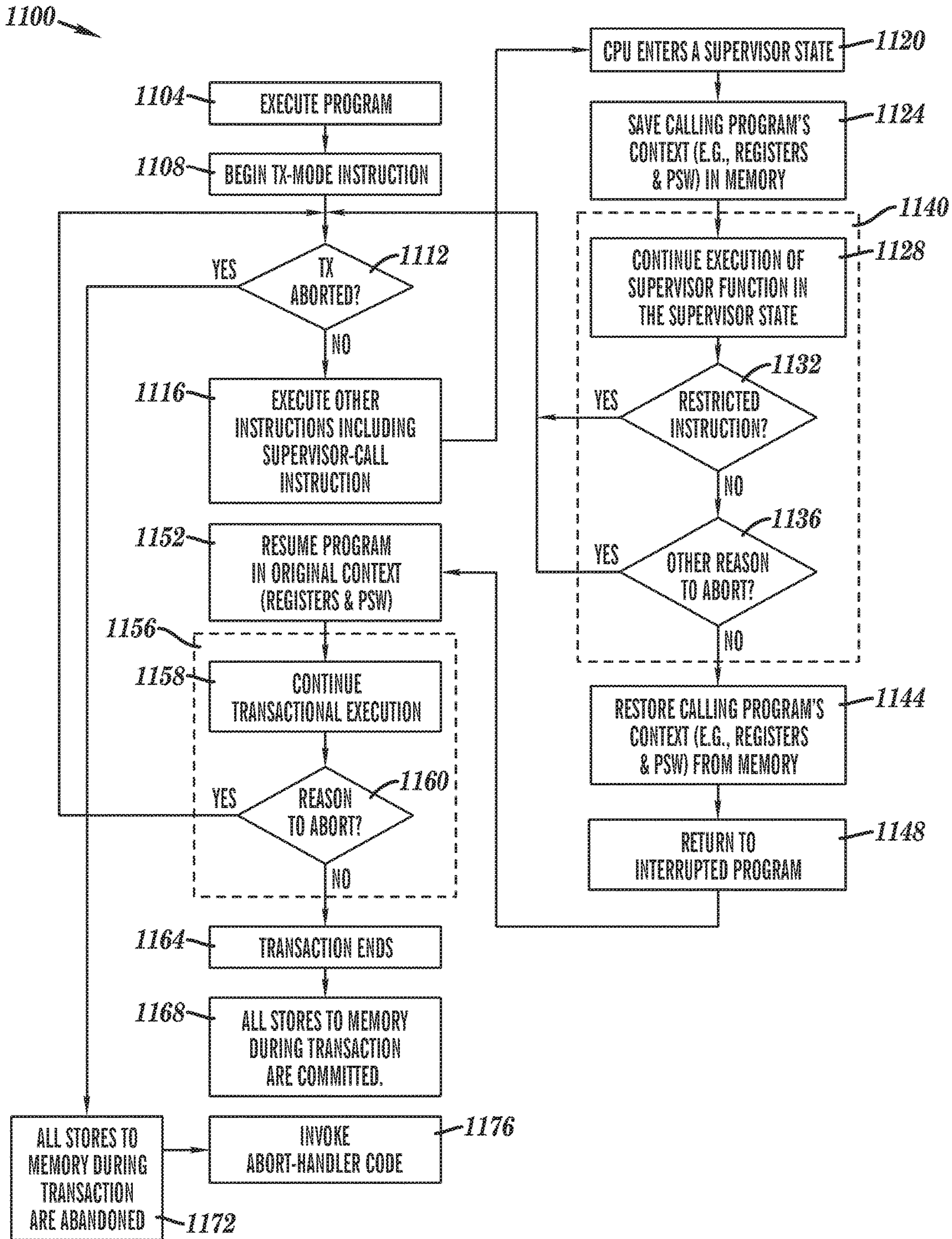


FIG. 9

1

**TRANSACTIONAL EXECUTION ENABLED
SUPERVISOR CALL INTERRUPTION
WHILE IN TX MODE**

CROSS REFERENCE TO RELATED
APPLICATIONS

This application is a continuation of U.S. application Ser. No. 14/316,894, filed Jun. 27, 2014.

BACKGROUND

The present disclosure relates generally to a computer implemented method for managing an interruption while a processor is in a transactional execution (TX) mode executing a transaction.

The number of central processing unit (CPU) cores on a chip and the number of CPU cores connected to a shared memory continues to grow significantly to support growing workload capacity demand. The increasing number of CPUs cooperating to process the same workloads puts a significant burden on software scalability; for example, shared queues or data-structures protected by traditional semaphores become hot spots and lead to sub-linear n-way scaling curves. Traditionally this has been countered by implementing finer-grained locking in software, and with lower latency/higher bandwidth interconnects in hardware. Implementing fine-grained locking to improve software scalability can be very complicated and error-prone, and at today's CPU frequencies, the latencies of hardware interconnects are limited by the physical dimension of the chips and systems, and by the speed of light.

Implementations of hardware transactional memory (HTM, or in this discussion, simply TM) have been introduced, wherein a group of instructions—called a transaction—operate in an atomic manner on a data structure in memory, as viewed by other central processing units (CPUs) and the I/O subsystem (atomic operation is also known as block concurrent or serialized in other literature). The transaction executes optimistically without obtaining a lock, but may need to abort and retry the transaction execution if an operation, of the executing transaction, on a memory location conflicts with another operation on the same memory location. Previously, software transactional memory implementations have been proposed to support software Transactional Memory (STM). However, hardware TM can provide improved performance aspects and ease of use over software TM.

Commonly owned U.S. Pat. No. 6,381,606 titled "Application Programming Interface for Creating Authorized Connections to a Database Management System", filed Jun. 28, 1999, is incorporated by reference, and teaches a method, apparatus, and article of manufacture for accessing a database management system. An application programming interface (API) provides a plurality of simplified procedures that allow an application program executed by the computer to access a database management system (DBMS) by creating an authorized connection between the application program and the DBMS. The application program invokes the simplified procedures of the API, the invoked procedures generate at least one supervisor call (SVC) interrupt that transfers control to an SVC Service Routine, and the SVC Service Routine connects to a Cross Coupling Facility (XCF) that interfaces to an Open Transaction Manager Access (OTMA) component of the DBMS.

Commonly owned U.S. Pat. No. 5,715,458 titled "Incorporation of Services Written in one Operating System Envi-

2

ronment into Another Operating System Environment", filed Feb. 13, 1995, is incorporated by reference, and teaches a data processing apparatus which comprises a processor and memory. The processor has a supervisor state including a first set of supervisor service routines for controlling the operation of the data processing apparatus. The memory has a first hash table accessible by the supervisor state for pointing to the first set of supervisor service routines and a second hash table also accessible by the supervisor state for pointing to a second set of supervisor service routines. The data processing apparatus provides table indicating means for indicating whether the first hash table or the second hash table is accessible by the supervisor state. The table indicating means is provided as an index into the first hash table. When this index is accessed, further code is executed which allows access to the second hash table. The operation of the data processing apparatus is controlled by a routine comprising the steps of testing to determine which hash table is to be used for calling the routine, using the determined hash table to find the address in the memory at which the code representing the called routine is resident, and executing the routine in the processor. In one embodiment of the '458 disclosure, the first step comprises testing the operating system environment in the processor. The '458 disclosure finds particular use in providing in a first operating system environment, such as MVS/ESA™, services written for a second operating system environment, such as VSE/ESA™.

SUMMARY

In an aspect according the disclosure, a computer implemented method for managing an interruption while a processor is executing a transaction in a transactional-execution (TX) mode. The method includes executing, by a processor initiated into a TX mode by the execution, a transaction in a program context. An interruption request can be detected, by the processor in the TX mode, for an interruption. Based on the interruption being a TX compatible routine, the processor accepts the interruption to execute the TX compatible routine in a supervisor context for changing supervisor resources. The method executes the TX compatible routine within the TX mode, returning to the program context to complete the execution of the transaction.

In another aspect according to the disclosure, a computer program product can manage an interruption while a processor is executing a transaction in a transactional-execution (TX) mode. The product comprises a computer readable storage medium readable by a processing circuit and storing instructions for execution by the processing circuit for performing a method comprising: executing, by a processor initiated into a TX mode by the executing, a transaction in a program context; detecting, by the processor in the TX mode, an interruption request for an interruption; based on the interruption being a TX compatible routine, accepting the interruption by the processor to execute the TX compatible routine in a supervisor context for changing supervisor resources; executing the TX compatible routine within the TX mode; and returning to the program context to complete execution of the transaction.

In another aspect according to the disclosure, a computer system can manage an interruption while a processor is executing a transaction in a transactional-execution (TX) mode. The computer system includes a memory, and a processor in communication with the memory, wherein the computer system is configured to perform a method. The method includes: executing, by a processor initiated into a TX mode by the execution, a transaction in a program

context; detecting, by the processor in the TX mode, an interruption request for an interruption; based on the interruption being a TX compatible routine, accepting the interruption by the processor to execute the TX compatible routine in a supervisor context for changing supervisor resources; executing the TX compatible routine within said TX mode; and returning to the program context to complete the execution of the transaction.

Additional features and advantages are realized through the techniques of the present disclosure. Other embodiments and aspects of the disclosure are described in detail herein and are considered a part of the claimed invention.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

One or more aspects of the present disclosure are particularly pointed out and distinctly claimed as examples in the claims at the conclusion of the specification. The foregoing and other objects, features, and advantages of the disclosure are apparent from the following detailed description taken in conjunction with the accompanying drawings in which:

FIG. 1 is a schematic block diagram of an example multiple processor (CPU)/core Transactional Memory environment according to an embodiment of the disclosure;

FIG. 2 is a schematic block diagram illustrating a transactional processor according to an embodiment of the disclosure;

FIG. 3 is a schematic block diagram of exemplary components of a transactional processor (CPU) shown in FIGS. 1 and 2, according to an embodiment of the disclosure;

FIG. 4 is a schematic block diagram of an illustrative multi-processor operating system in a symmetric I/O (input/output) mode;

FIG. 5 is a functional block diagram of a generic computer system having similar components as the multi processor systems shown in FIGS. 1-4 according to an embodiment of the disclosure;

FIG. 6 is a flow chart of a method for managing an interruption while a processor is executing a transaction in a transactional-execution (TX) mode, according to an embodiment of the disclosure;

FIG. 7 is a continuation of the flow chart shown in FIG. 6, depicting another embodiment according to the disclosure;

FIG. 8 is a schematic block diagram of an illustrative multi-processor operating system in a symmetric I/O (input/output) mode; and

FIG. 9 is a flow chart of a method for managing an interruption for a processor in a transactional execution mode, according to an embodiment of the disclosure.

DETAILED DESCRIPTION

Historically, a computer system or processor had only a single processor (aka processing unit or central processing unit). The processor included an instruction processing unit (IPU), a branch unit, a memory control unit and the like. Such processors were capable of executing a single thread of a program at a time. Operating systems were developed that could time-share a processor by dispatching a program to be executed on the processor for a period of time, and then dispatching another program to be executed on the processor for another period of time. As technology evolved, memory subsystem caches were often added to the processor as well as complex dynamic address translation including transla-

tion lookaside buffers (TLBs). The IPU itself was often referred to as a processor. As technology continued to evolve, an entire processor could be packaged in a single semiconductor chip or die, such a processor was referred to as a microprocessor. Then processors were developed that incorporated multiple IPUs, such processors were often referred to as multi-processors. Each such processor of a multi-processor computer system (processor) may include individual or shared caches, memory interfaces, system bus, address translation mechanism and the like. Virtual machine and instruction set architecture (ISA) emulators added a layer of software to a processor, that provided the virtual machine with multiple “virtual processors” (aka processors) by time-slice usage of a single IPU in a single hardware processor. As technology further evolved, multi-threaded processors were developed, enabling a single hardware processor having a single multi-thread IPU to provide a capability of simultaneously executing threads of different programs, thus each thread of a multi-threaded processor appeared to the operating system as a processor. As technology further evolved, it was possible to put multiple processors (each having an IPU) on a single semiconductor chip or die. These processors were referred to processor cores or just cores. Thus the terms such as processor, central processing unit, processing unit, microprocessor, core, processor core, processor thread, and thread for example are often used interchangeably. Aspects of embodiments herein may be practiced by any or all processors including those shown supra, without departing from the teachings herein. Wherein the term “thread” or “processor thread” is used herein, it is expected that particular advantage of the embodiment may be had in a processor thread implementation.

Transaction Execution in Intel® Based Embodiments

In “Intel® Architecture Instruction Set Extensions Programming Reference” 319433-012A, Feb. 2012, incorporated by reference, Chapter 8 teaches, in part, that multi-threaded applications may take advantage of increasing numbers of CPU cores to achieve higher performance. However, the writing of multi-threaded applications requires programmers to understand and take into account data sharing among the multiple threads. Access to shared data typically requires synchronization mechanisms. These synchronization mechanisms are used to ensure that multiple threads update shared data by serializing operations that are applied to the shared data, often through the use of a critical section that is protected by a lock. Since serialization limits concurrency, programmers try to limit the overhead due to synchronization.

Intel® Transactional Synchronization Extensions (Intel® TSX) allow a processor to dynamically determine whether threads need to be serialized through lock-protected critical sections, and to perform that serialization only when required. This allows the processor to expose and exploit concurrency that is hidden in an application because of dynamically unnecessary synchronization.

With Intel TSX, programmer-specified code regions (also referred to as “transactional regions” or just “transactions”) are executed transactionally. If the transactional execution completes successfully, then all memory operations performed within the transactional region will appear to have occurred instantaneously when viewed from other processors. A processor makes the memory operations of the executed transaction, performed within the transactional region, visible to other processors only when a successful

commit occurs, i.e., when the transaction successfully completes execution. This process is often referred to as an atomic commit.

Intel TSX provides two software interfaces to specify regions of code for transactional execution. Hardware Lock Elision (HLE) is a legacy compatible instruction set extension (comprising the XACQUIRE and XRELEASE prefixes) to specify transactional regions. Restricted Transactional Memory (RTM) is a new instruction set interface (comprising the XBEGIN, XEND, and XABORT instructions) for programmers to define transactional regions in a more flexible manner than that possible with HLE. HLE is for programmers who prefer the backward compatibility of the conventional mutual exclusion programming model and would like to run HLE-enabled software on legacy hardware but would also like to take advantage of the new lock elision capabilities on hardware with HLE support. RTM is for programmers who prefer a flexible interface to the transactional execution hardware. In addition, Intel TSX also provides an XTEST instruction. This instruction allows software to query whether the logical processor is transactionally executing in a transactional region identified by either HLE or RTM.

Since a successful transactional execution ensures an atomic commit, the processor executes the code region optimistically without explicit synchronization. If synchronization was unnecessary for that specific execution, execution can commit without any cross-thread serialization. If the processor cannot commit atomically, then the optimistic execution fails. When this happens, the processor will roll back the execution, a process referred to as a transactional abort. On a transactional abort, the processor will discard all updates performed in the memory region used by the transaction, restore architectural state to appear as if the optimistic execution never occurred, and resume execution non-transactionally.

A processor can perform a transactional abort for numerous reasons. A primary reason to abort a transaction is due to conflicting memory accesses between the transactionally executing logical processor and another logical processor. Such conflicting memory accesses may prevent a successful transactional execution. Memory addresses read from within a transactional region constitute the read-set of the transactional region and addresses written to within the transactional region constitute the write-set of the transactional region. Intel TSX maintains the read- and write-sets at the granularity of a cache line. A conflicting memory access occurs if another logical processor either reads a location that is part of the transactional region's write-set or writes a location that is a part of either the read- or write-set of the transactional region. A conflicting access typically means that serialization is required for this code region. Since Intel TSX detects data conflicts at the granularity of a cache line, unrelated data locations placed in the same cache line will be detected as conflicts that result in transactional aborts. Transactional aborts may also occur due to limited transactional resources. For example, the amount of data accessed in the region may exceed an implementation-specific capacity. Additionally, some instructions and system events may cause transactional aborts. Frequent transactional aborts result in wasted cycles and increased inefficiency.

Hardware Lock Elision

Hardware Lock Elision (HLE) provides a legacy compatible instruction set interface for programmers to use transactional execution. HLE provides two new instruction prefix hints: XACQUIRE and XRELEASE.

With HLE, a programmer adds the XACQUIRE prefix to the front of the instruction that is used to acquire the lock that is protecting the critical section. The processor treats the prefix as a hint to elide the write associated with the lock acquire operation. Even though the lock acquire has an associated write operation to the lock, the processor does not add the address of the lock to the transactional region's write-set nor does it issue any write requests to the lock. Instead, the address of the lock is added to the read-set. The logical processor enters transactional execution. If the lock was available before the XACQUIRE prefixed instruction, then all other processors will continue to see the lock as available afterwards. Since the transactionally executing logical processor neither added the address of the lock to its write-set nor performed externally visible write operations to the lock, other logical processors can read the lock without causing a data conflict. This allows other logical processors to also enter and concurrently execute the critical section protected by the lock. The processor automatically detects any data conflicts that occur during the transactional execution and will perform a transactional abort if necessary.

Even though the eliding processor did not perform any external write operations to the lock, the hardware ensures program order of operations on the lock. If the eliding processor itself reads the value of the lock in the critical section, it will appear as if the processor had acquired the lock, i.e. the read will return the non-elided value. This behavior allows an HLE execution to be functionally equivalent to an execution without the HLE prefixes.

An XRELEASE prefix can be added in front of an instruction that is used to release the lock protecting a critical section. Releasing the lock involves a write to the lock. If the instruction is to restore the value of the lock to the value the lock had prior to the XACQUIRE prefixed lock acquire operation on the same lock, then the processor elides the external write request associated with the release of the lock and does not add the address of the lock to the write-set. The processor then attempts to commit the transactional execution.

With HLE, if multiple threads execute critical sections protected by the same lock but they do not perform any conflicting operations on each other's data, then the threads can execute concurrently and without serialization. Even though the software uses lock acquisition operations on a common lock, the hardware recognizes this, elides the lock, and executes the critical sections on the two threads without requiring any communication through the lock—if such communication was dynamically unnecessary.

If the processor is unable to execute the region transactionally, then the processor will execute the region non-transactionally and without elision. HLE enabled software has the same forward progress guarantees as the underlying non-HLE lock-based execution. For successful HLE execution, the lock and the critical section code must follow certain guidelines. These guidelines only affect performance; and failure to follow these guidelines will not result in a functional failure. Hardware without HLE support will ignore the XACQUIRE and XRELEASE prefix hints and will not perform any elision since these prefixes correspond to the REPNE/REPE IA-32 prefixes which are ignored on the instructions where XACQUIRE and XRELEASE are valid. Importantly, HLE is compatible with the existing lock-based programming model. Improper use of hints will not cause functional bugs though it may expose latent bugs already in the code.

Restricted Transactional Memory (RTM) provides a flexible software interface for transactional execution. RTM

provides three new instructions—XBEGIN, XEND, and XABORT—for programmers to start, commit, and abort a transactional execution.

The programmer uses the XBEGIN instruction to specify the start of a transactional code region and the XEND instruction to specify the end of the transactional code region. If the RTM region could not be successfully executed transactionally, then the XBEGIN instruction takes an operand that provides a relative offset to the fallback instruction address.

A processor may abort RTM transactional execution for many reasons. In many instances, the hardware automatically detects transactional abort conditions and restarts execution from the fallback instruction address with the architectural state corresponding to that present at the start of the XBEGIN instruction and the EAX register updated to describe the abort status.

The XABORT instruction allows programmers to abort the execution of an RTM region explicitly. The XABORT instruction takes an 8-bit immediate argument that is loaded into the EAX register and will thus be available to software following an RTM abort. RTM instructions do not have any data memory location associated with them. While the hardware provides no guarantees as to whether an RTM region will ever successfully commit transactionally, most transactions that follow the recommended guidelines are expected to successfully commit transactionally. However, programmers must always provide an alternative code sequence in the fallback path to guarantee forward progress. This may be as simple as acquiring a lock and executing the specified code region non-transactionally. Further, a transaction that always aborts on a given implementation may complete transactionally on a future implementation. Therefore, programmers must ensure the code paths for the transactional region and the alternative code sequence are functionally tested.

Detection of HLE Support

A processor supports HLE execution if CPUID.07H.EBX.HLE [bit 4]=1. However, an application can use the HLE prefixes (XACQUIRE and XRELEASE) without checking whether the processor supports HLE. Processors without HLE support ignore these prefixes and will execute the code without entering transactional execution.

Detection of RTM Support

A processor supports RTM execution if CPUID.07H.EBX.RTM [bit 11]=1. An application must check if the processor supports RTM before it uses the RTM instructions (XBEGIN, XEND, XABORT). These instructions will generate a #UD exception when used on a processor that does not support RTM.

Detection of XTEST Instruction

A processor supports the XTEST instruction if it supports either HLE or RTM. An application must check either of these feature flags before using the XTEST instruction. This instruction will generate a #UD exception when used on a processor that does not support either HLE or RTM.

Querying Transactional Execution Status

The XTEST instruction can be used to determine the transactional status of a transactional region specified by HLE or RTM. Note, while the HLE prefixes are ignored on processors that do not support HLE, the XTEST instruction will generate a #UD exception when used on processors that do not support either HLE or RTM.

Requirements for HLE Locks

For HLE execution to successfully commit transactionally, the lock must satisfy certain properties and access to the lock must follow certain guidelines.

An XRELEASE prefixed instruction must restore the value of the elided lock to the value it had before the lock acquisition. This allows hardware to safely elide locks by not adding them to the write-set. The data size and data address of the lock release (XRELEASE prefixed) instruction must match that of the lock acquire (XACQUIRE prefixed) and the lock must not cross a cache line boundary.

Software should not write to the elided lock inside a transactional HLE region with any instruction other than an XRELEASE prefixed instruction, otherwise such a write may cause a transactional abort. In addition, recursive locks (where a thread acquires the same lock multiple times without first releasing the lock) may also cause a transactional abort. Note that software can observe the result of the elided lock acquire inside the critical section. Such a read operation will return the value of the write to the lock.

The processor automatically detects violations to these guidelines, and safely transitions to a non-transactional execution without elision. Since Intel TSX detects conflicts at the granularity of a cache line, writes to data collocated on the same cache line as the elided lock may be detected as data conflicts by other logical processors eliding the same lock.

Transactional Nesting

Both HLE and RTM support nested transactional regions. However, a transactional abort restores state to the operation that started transactional execution: either the outermost XACQUIRE prefixed HLE eligible instruction or the outermost XBEGIN instruction. The processor treats all nested transactions as one transaction.

HLE Nesting and Elision

Programmers can nest HLE regions up to an implementation specific depth of MAX_HLE_NEST_COUNT. Each logical processor tracks the nesting count internally but this count is not available to software. An XACQUIRE prefixed HLE-eligible instruction increments the nesting count, and an XRELEASE prefixed HLE-eligible instruction decrements it. The logical processor enters transactional execution when the nesting count goes from zero to one. The logical processor attempts to commit only when the nesting count becomes zero. A transactional abort may occur if the nesting count exceeds MAX_HLE_NEST_COUNT.

In addition to supporting nested HLE regions, the processor can also elide multiple nested locks. The processor tracks a lock for elision beginning with the XACQUIRE prefixed HLE eligible instruction for that lock and ending with the XRELEASE prefixed HLE eligible instruction for that same lock. The processor can, at any one time, track up to a MAX_HLE_ELIDED_LOCKS number of locks. For example, if the implementation supports a MAX_HLE_ELIDED_LOCKS value of two and if the programmer nests three HLE identified critical sections (by performing XACQUIRE prefixed HLE eligible instructions on three distinct locks without performing an intervening XRELEASE prefixed HLE eligible instruction on any one of the locks), then the first two locks will be elided, but the third won't be elided (but will be added to the transaction's write set). However, the execution will still continue transactionally. Once an XRELEASE for one of the two elided locks is encountered, a subsequent lock acquired through the XACQUIRE prefixed HLE eligible instruction will be elided.

The processor attempts to commit the HLE execution when all elided XACQUIRE and XRELEASE pairs have been matched, the nesting count goes to zero, and the locks

have satisfied requirements. If execution cannot commit atomically, then execution transitions to a non-transactional execution without elision as if the first instruction did not have an XACQUIRE prefix.

RTM Nesting

Programmers can nest RTM regions up to an implementation specific MAX_RT_M_NEST_COUNT. The logical processor tracks the nesting count internally but this count is not available to software. An XBEGIN instruction increments the nesting count, and an XEND instruction decrements the nesting count. The logical processor attempts to commit only if the nesting count becomes zero. A transactional abort occurs if the nesting count exceeds MAX_RT_M_NEST_COUNT.

Nesting HLE and RTM

HLE and RTM provide two alternative software interfaces to a common transactional execution capability. Transactional processing behavior is implementation specific when HLE and RTM are nested together, e.g., HLE is inside RTM or RTM is inside HLE. However, in all cases, the implementation will maintain HLE and RTM semantics. An implementation may choose to ignore HLE hints when used inside RTM regions, and may cause a transactional abort when RTM instructions are used inside HLE regions. In the latter case, the transition from transactional to non-transactional execution occurs seamlessly since the processor will re-execute the HLE region without actually doing elision, and then execute the RTM instructions.

Abort Status Definition

RTM uses the EAX register to communicate abort status to software. Following an RTM abort the EAX register has the following definition.

TABLE 1

RTM Abort Status Definition	
EAX Register Bit Position	Meaning
0	Set if abort caused by XABORT instruction
1	If set, the transaction may succeed on retry, this bit is always clear if bit 0 is set
2	Set if another logical processor conflicted with a memory address that was part of the transaction that aborted
3	Set if an internal buffer overflowed
4	Set if a debug breakpoint was hit
5	Set if an abort occurred during execution of a nested transaction
23:6	Reserved
31-24	XABORT argument (only valid if bit 0 set, otherwise reserved)

The EAX abort status for RTM only provides causes for aborts. It does not by itself encode whether an abort or commit occurred for the RTM region. The value of EAX can be 0 following an RTM abort. For example, a CPUID instruction when used inside an RTM region causes a transactional abort and may not satisfy the requirements for setting any of the EAX bits. This may result in an EAX value of 0.

RTM Memory Ordering

A successful RTM commit causes all memory operations in the RTM region to appear to execute atomically. A successfully committed RTM region consisting of an XBEGIN followed by an XEND, even with no memory operations in the RTM region, has the same ordering semantics as a LOCK prefixed instruction.

The XBEGIN instruction does not have fencing semantics. However, if an RTM execution aborts, then all memory

updates from within the RTM region are discarded and are not made visible to any other logical processor.

RTM-Enabled Debugger Support

By default, any debug exception inside an RTM region will cause a transactional abort and will redirect control flow to the fallback instruction address with architectural state recovered and bit 4 in EAX set. However, to allow software debuggers to intercept execution on debug exceptions, the RTM architecture provides additional capability.

If bit 11 of DR7 and bit 15 of the IA32_DEBUGCTL_MSR are both 1, any RTM abort due to a debug exception (#DB) or breakpoint exception (#BP) causes execution to roll back and restart from the XBEGIN instruction instead of the fallback address. In this scenario, the EAX register will also be restored back to the point of the XBEGIN instruction.

Programming Considerations

Typical programmer-identified regions are expected to transactionally execute and commit successfully. However, Intel TSX does not provide any such guarantee. A transactional execution may abort for many reasons. To take full advantage of the transactional capabilities, programmers should follow certain guidelines to increase the probability of their transactional execution committing successfully.

This section discusses various events that may cause transactional aborts. The architecture ensures that updates performed within a transaction that subsequently aborts execution will never become visible. Only committed transactional executions initiate an update to the architectural state. Transactional aborts never cause functional failures and only affect performance.

Instruction Based Considerations

Programmers can use any instruction safely inside a transaction (HLE or RTM) and can use transactions at any privilege level. However, some instructions will always abort the transactional execution and cause execution to seamlessly and safely transition to a non-transactional path.

Intel TSX allows for most common instructions to be used inside transactions without causing aborts. The following operations inside a transaction do not typically cause an abort:

Operations on the instruction pointer register, general purpose registers (GPRs) and the status flags (CF, OF, SF, PF, AF, and ZF); and

Operations on XMM and YMM registers and the MXCSR register.

However, programmers must be careful when intermixing SSE and AVX operations inside a transactional region. Intermixing SSE instructions accessing XMM registers and AVX instructions accessing YMM registers may cause transactions to abort. Programmers may use REP/REPNE prefixed string operations inside transactions. However, long strings may cause aborts. Further, the use of CLD and STD instructions may cause aborts if they change the value of the DF flag. However, if DF is 1, the STD instruction will not cause an abort. Similarly, if DF is 0, then the CLD instruction will not cause an abort.

Instructions not enumerated here as causing abort when used inside a transaction will typically not cause a transaction to abort (examples include but are not limited to MFENCE, LFENCE, SFENCE, RDTSC, RDTSCP, etc.).

The following instructions will abort transactional execution on any implementation:

XABORT
CPUID
PAUSE

In addition, in some implementations, the following instructions may always cause transactional aborts. These instructions are not expected to be commonly used inside typical transactional regions. However, programmers must not rely on these instructions to force a transactional abort, since whether they cause transactional aborts is implementation dependent.

Operations on X87 and MMX architecture state. This includes all MMX and X87 instructions, including the FXRSTOR and FXSAVE instructions.

Update to non-status portion of EFLAGS: CLI, STI, POPFD, POPFQ, and CLTS.

Instructions that update segment registers, debug registers and/or control registers: MOV to DS/ES/FS/GS/SS, POP DS/ES/FS/GS/SS, LDS, LES, LFS, LGS, LSS, SWAPGS, WRFSBASE, WRGSBASE, LGDT, SGDT, LIDT, SIDT, LLDT, SLDT, LTR, STR, Far CALL, Far JMP, Far RET, IRET, MOV to DRx, MOV to CR0/CR2/CR3/CR4/CR8 and LMSW.

Ring transitions: SYSENTER, SYSCALL, SYSEXIT, and SYSRET.

TLB and Cacheability control: CLFLUSH, INVD, WBINVD, INVLPG, INVPCID, and memory instructions with a non-temporal hint (MOVNTDQA, MOVNTDQ, MOVNTI, MOVNTPD, MOVNTPS, and MOVNTQ).

Processor state save: XSAVE, XSAVEOPT, and XRSTOR.

Interrupts: INTn, INTO.

IO: IN, INS, REP INS, OUT, OUTS, REP OUTS and their variants.

VMX: VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON, INVEPT, and INVVPID.

SMX: GETSEC.

UD2, RSM, RDMSR, WRMSR, HLT, MONITOR, MWAIT, XSETBV, VZEROUPPER, MASKMOVQ, and V/MASKMOVDQU.

Runtime Considerations

In addition to the instruction-based considerations, runtime events may cause transactional execution to abort. These may be due to data access patterns or micro-architectural implementation features. The following list is not a comprehensive discussion of all abort causes.

Any fault or trap in a transaction that must be exposed to software will be suppressed. Transactional execution will abort and execution will transition to a non-transactional execution, as if the fault or trap had never occurred. If an exception is not masked, then that un-masked exception will result in a transactional abort and the state will appear as if the exception had never occurred.

Synchronous exception events (#DE, #OF, #NP, #SS, #GP, #BR, #UD, #AC, #XF, #PF, #NM, #TS, #MF, #DB, #BP/INT3) that occur during transactional execution may cause an execution not to commit transactionally, and require a non-transactional execution. These events are suppressed as if they had never occurred. With HLE, since the non-transactional code path is identical to the transactional code path, these events will typically re-appear when the instruction that caused the exception is re-executed non-transactionally, causing the associated synchronous events to be delivered appropriately in the non-transactional execution. Asynchronous events (NMI, SMI, INTR, IPI, PMI, etc.) occurring during transactional execution may cause the transactional execution to abort and transition to a

non-transactional execution. The asynchronous events will be pended and handled after the transactional abort is processed.

Transactions only support write-back cacheable memory type operations. A transaction may always abort if the transaction includes operations on any other memory type. This includes instruction fetches to UC memory type.

Memory accesses within a transactional region may require the processor to set the Accessed and Dirty flags of the referenced page table entry. The behavior of how the processor handles this is implementation specific. Some implementations may allow the updates to these flags to become externally visible even if the transactional region subsequently aborts. Some Intel TSX implementations may choose to abort the transactional execution if these flags need to be updated. Further, a processor's page-table walk may generate accesses to its own transactionally written but uncommitted state. Some Intel TSX implementations may choose to abort the execution of a transactional region in such situations. Regardless, the architecture ensures that, if the transactional region aborts, then the transactionally written state will not be made architecturally visible through the behavior of structures such as TLBs.

Executing self-modifying code transactionally may also cause transactional aborts. Programmers must continue to follow the Intel recommended guidelines for writing self-modifying and cross-modifying code even when employing HLE and RTM. While an implementation of RTM and HLE will typically provide sufficient resources for executing common transactional regions, implementation constraints and excessive sizes for transactional regions may cause a transactional execution to abort and transition to a non-transactional execution. The architecture provides no guarantee of the amount of resources available to do transactional execution and does not guarantee that a transactional execution will ever succeed.

Conflicting requests to a cache line accessed within a transactional region may prevent the transaction from executing successfully. For example, if logical processor P0 reads line A in a transactional region and another logical processor P1 writes line A (either inside or outside a transactional region) then logical processor P0 may abort if logical processor P1's write interferes with processor P0's ability to execute transactionally.

Similarly, if P0 writes line A in a transactional region and P1 reads or writes line A (either inside or outside a transactional region), then P0 may abort if P1's access to line A interferes with P0's ability to execute transactionally. In addition, other coherence traffic may at times appear as conflicting requests and may cause aborts. While these false conflicts may happen, they are expected to be uncommon. The conflict resolution policy to determine whether P0 or P1 aborts in the above scenarios is implementation specific.

Generic Transaction Execution embodiments:

According to "ARCHITECTURES FOR TRANSACTIONAL MEMORY", a dissertation submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy, by Austen McDonald, Jun. 2009, incorporated by reference, fundamentally, there are three mechanisms needed to implement an atomic and isolated transactional region: versioning, conflict detection, and contention management.

To make a transactional code region appear atomic, all the modifications performed by that transactional code region must be stored and kept isolated from other transactions

until commit time. The system does this by implementing a versioning policy. Two versioning paradigms exist: eager and lazy. An eager versioning system stores newly generated transactional values in place and stores previous memory values on the side, in what is called an undo-log. A lazy

versioning system stores new values temporarily in what is called a write buffer, copying them to memory only on commit. In either system, the cache is used to optimize storage of new versions. To ensure that transactions appear to be performed atomically, conflicts must be detected and resolved. The two systems, i.e., the eager and lazy versioning systems, detect conflicts by implementing a conflict detection policy, either optimistic or pessimistic. An optimistic system executes transactions in parallel, checking for conflicts only when a transaction commits. A pessimistic system checks for conflicts at each load and store. Similar to versioning, conflict detection also uses the cache, marking each line as either part of the read-set, part of the write-set, or both. The two systems resolve conflicts by implementing a contention management policy. Many contention management policies exist, some are more appropriate for optimistic conflict detection and some are more appropriate for pessimistic. Described below are some example policies.

Since each transactional memory (TM) system needs both versioning detection and conflict detection, these options give rise to four distinct TM designs: Eager-Pessimistic (EP), Eager-Optimistic (EO), Lazy-Pessimistic (LP), and Lazy-Optimistic (LO). Table 2 briefly describes all four distinct TM designs.

FIGS. 1 and 2 depict an example of a multicore TM environment. FIG. 1 shows many TM-enabled CPUs (CPU1 114a, CPU2 114b, etc.) on one die 100, connected with an interconnect 122, under management of an interconnect control 120a, 120b. Each CPU 114a, 114b (also known as a Processor) may have a split cache consisting of an Instruction Cache 116a, 116b for caching instructions from memory to be executed and a Data Cache 118a, 118b with TM support for caching data (operands) of memory locations to be operated on by the CPU 114a, 114b (in FIG. 1, each CPU 114a, 114b and its associated caches are referenced as 112a, 112b). In an implementation, caches of multiple dies 100 are interconnected to support cache coherency between the caches of the multiple dies 100. In an implementation, a single cache, rather than the split cache is employed holding both instructions and data. In implementations, the CPU caches are one level of caching in a hierarchical cache structure. For example each die 100 may employ a shared cache 124 to be shared amongst all the CPUs on the die 100. In another implementation, each die

may have access to a shared cache 124, shared amongst all the processors of all the dies 100.

FIG. 2 shows the details of an example transactional CPU 114, including additions to support TM. The transactional CPU (processor) 114 may include hardware for supporting Register Checkpoints 126 and special TM Registers 128. The transactional CPU cache may have the MESI bits 130, Tags 140 and Data 142 of a conventional cache but also, for example, R bits 132 showing a line has been read by the CPU 114 while executing a transaction and W bits 138 showing a line has been written-to by the CPU 114 while executing a transaction.

A key detail for programmers in any TM system is how non-transactional accesses interact with transactions. By design, transactional accesses are screened from each other using the mechanisms above. However, the interaction between a regular, non-transactional load with a transaction containing a new value for that address must still be considered. In addition, the interaction between a non-transactional store with a transaction that has read that address must also be explored. These are issues of the database concept isolation.

A TM system is said to implement strong isolation, sometimes called strong atomicity, when every non-transactional load and store acts like an atomic transaction. Therefore, non-transactional loads cannot see uncommitted data and non-transactional stores cause atomicity violations in any transactions that have read that address. A system where this is not the case is said to implement weak isolation, sometimes called weak atomicity.

Strong isolation is often more desirable than weak isolation due to the relative ease of conceptualization and implementation of strong isolation. Additionally, if a programmer has forgotten to surround some shared memory references with transactions, causing bugs, then with strong isolation, the programmer will often detect that oversight using a simple debug interface because the programmer will see a non-transactional region causing atomicity violations. Also, programs written in one model may work differently on another model.

Further, strong isolation is often easier to support in hardware TM than weak isolation. With strong isolation, since the coherence protocol already manages load and store communication between processors, transactions can detect non-transactional loads and stores and act appropriately. To implement strong isolation in software Transactional Memory (TM), non-transactional code must be modified to include read- and write-barriers; potentially crippling performance. Although great effort has been expended to remove many un-needed barriers, such techniques are often complex and performance is typically far lower than that of HTMs.

TABLE 2

Transactional Memory Design Space			
VERSIONING			
		Lazy	Eager
CONFLICT DETECTION	Optimistic	Storing updates in a write buffer; detecting conflicts at commit time.	Not practical: waiting to update memory until commit time but detecting conflicts at access time guarantees wasted work and provides no advantage
	Pessimistic	Storing updates in a write buffer; detecting conflicts at access time.	Updating memory, keeping old values in undo log; detecting conflicts at access time.

Table 2 illustrates the fundamental design space of transactional memory (versioning and conflict detection).

Eager-Pessimistic (EP)

This first TM design described below is known as Eager-Pessimistic. An EP system stores its write-set “in place” (hence the name “eager”) and, to support rollback, stores the old values of overwritten lines in an “undo log”. Processors use the **W 138** and **R 132** cache bits to track read and write-sets and detect conflicts when receiving snooped load requests. Perhaps the most notable examples of EP systems in known literature are LogTM and UTM.

Beginning a transaction in an EP system is much like beginning a transaction in other systems: `tm_begin()` takes a register checkpoint, and initializes any status registers. An EP system also requires initializing the undo log, the details of which are dependent on the log format, but often involve initializing a log base pointer to a region of pre-allocated, thread-private memory, and clearing a log bounds register.

Versioning: In EP, due to the way eager versioning is designed to function, the MESI state transitions (cache line indicators corresponding to Modified, Exclusive, Shared, and Invalid code states) are left mostly unchanged. Outside of a transaction, the MESI state transitions are left completely unchanged. When reading a line inside a transaction, the standard coherence transitions apply (S (Shared)→S, I (Invalid)→S, or I→E (Exclusive)), issuing a load miss as needed, but the **R 132** bit is also set. Likewise, writing a line applies the standard transitions (S→M, E→I, I→M), issuing a miss as needed, but also sets the **W 138** (Written) bit. The first time a line is written, the old version of the entire line is loaded then written to the undo log to preserve it in case the current transaction aborts. The newly written data is then stored “in-place,” over the old data.

Conflict Detection: Pessimistic conflict detection uses coherence messages exchanged on misses, or upgrades, to look for conflicts between transactions. When a read miss occurs within a transaction, other processors receive a load request; but they ignore the request if they do not have the needed line. If the other processors have the needed line non-speculatively or have the line **R 132** (Read), they downgrade that line to S, and in certain cases issue a cache-to-cache transfer if they have the line in MESI’s **130** M or E state. However, if the cache has the line **W 138**, then a conflict is detected between the two transactions and additional action(s) must be taken.

Similarly, when a transaction seeks to upgrade a line from shared to modified (on a first write), the transaction issues an exclusive load request, which is also used to detect conflicts. If a receiving cache has the line non-speculatively, then the line is invalidated, and in certain cases a cache-to-cache transfer (M or E states) is issued. But, if the line is **R 132** or **W 138**, a conflict is detected.

Validation: Because conflict detection is performed on every load, a transaction always has exclusive access to its own write-set. Therefore, validation does not require any additional work.

Commit: Since eager versioning stores the new version of data items in place, the commit process simply clears the **W 138** and **R 132** bits and discards the undo log.

Abort: When a transaction rolls back, the original version of each cache line in the undo log must be restored, a process called “unrolling” or “applying” the log. This is done during `tm_discard()` and must be atomic with regard to other transactions. Specifically, the write-set must still be used to detect conflicts: this transaction has the only correct version of lines in its undo log, and requesting transactions must

wait for the correct version to be restored from that log. Such a log can be applied using a hardware state machine or software abort handler.

Eager-Pessimistic has the characteristics of: Commit is simple and since it is in-place, very fast. Similarly, validation is a no-op. Pessimistic conflict detection detects conflicts early, thereby reducing the number of “doomed” transactions. For example, if two transactions are involved in a Write-After-Read dependency, then that dependency is detected immediately in pessimistic conflict detection. However, in optimistic conflict detection such conflicts are not detected until the writer commits.

Eager-Pessimistic also has the characteristics of: As described above, the first time a cache line is written, the old value must be written to the log, incurring extra cache accesses. Aborts are expensive as they require undoing the log. For each cache line in the log, a load must be issued, perhaps going as far as main memory before continuing to the next line. Pessimistic conflict detection also prevents certain serializable schedules from existing.

Additionally, because conflicts are handled as they occur, there is a potential for livelock and careful contention management mechanisms must be employed to guarantee forward progress.

Lazy-Optimistic (LO)

Another popular TM design is Lazy-Optimistic (LO), which stores its write-set in a “write buffer” or “redo log” and detects conflicts at commit time (still using the **R 132** and **W 138** bits).

Versioning: Just as in the EP system, the MESI protocol of the LO design is enforced outside of the transactions. Once inside a transaction, reading a line incurs the standard MESI transitions but also sets the **R 132** bit. Likewise, writing a line sets the **W 138** bit of the line, but handling the MESI transitions of the LO design is different from that of the EP design. First, with lazy versioning, the new versions of written data are stored in the cache hierarchy until commit while other transactions have access to old versions available in memory or other caches. To make available the old versions, dirty lines (M lines) must be evicted when first written by a transaction. Second, no upgrade misses are needed because of the optimistic conflict detection feature: if a transaction has a line in the S state, it can simply write to it and upgrade that line to an M state without communicating the changes with other transactions because conflict detection is done at commit time.

Conflict Detection and Validation: To validate a transaction and detect conflicts, LO communicates the addresses of speculatively modified lines to other transactions only when it is preparing to commit. On validation, the processor sends one, potentially large, network packet containing all the addresses in the write-set. Data is not sent, but left in the cache of the committer and marked dirty (M). To build this packet without searching the cache for lines marked W, a simple bit vector is used, called a “store buffer,” with one bit per cache line to track these speculatively modified lines. Other transactions use this address packet to detect conflicts: if an address is found in the cache and the **R 132** and/or **W 138** bits are set, then a conflict is initiated. If the line is found but neither **R 132** nor **W 138** is set, then the line is simply invalidated, which is similar to processing an exclusive load.

To support transaction atomicity, these address packets must be handled atomically, i.e., no two address packets may exist at once with the same addresses. In an LO system, this can be achieved by simply acquiring a global commit token before sending the address packet. However, a two-phase commit scheme could be employed by first sending out the

address packet, collecting responses, enforcing an ordering protocol (perhaps oldest transaction first), and committing once all responses are satisfactory.

Commit: Once validation has occurred, commit needs no special treatment: simply clear W **138** and R **132** bits and the store buffer. The transaction's writes are already marked dirty in the cache and other caches' copies of these lines have been invalidated via the address packet. Other processors can then access the committed data through the regular coherence protocol.

Abort: Rollback is equally easy: because the write-set is contained within the local caches, these lines can be invalidated, then clear W **138** and R **132** bits and the store buffer. The store buffer allows W lines to be found to invalidate without the need to search the cache.

Lazy-Optimistic has the characteristics of: Aborts are very fast, requiring no additional loads or stores and making only local changes. More serializable schedules can exist than found in EP, which allows an LO system to more aggressively speculate that transactions are independent, which can yield higher performance. Finally, the late detection of conflicts can increase the likelihood of forward progress.

Lazy-Optimistic also has the characteristics of: Validation takes global communication time proportional to size of write set. Doomed transactions can waste work since conflicts are detected only at commit time.

Lazy-Pessimistic (LP)

Lazy-Pessimistic (LP) represents a third TM design option, sitting somewhere between EP and LO: storing newly written lines in a write buffer but detecting conflicts on a per access basis.

Versioning: Versioning is similar but not identical to that of LO: reading a line sets its R bit **132**, writing a line sets its W bit **138**, and a store buffer is used to track W lines in the cache. Also, dirty (M) lines must be evicted when first written by a transaction, just as in LO. However, since conflict detection is pessimistic, load exclusives must be performed when upgrading a transactional line from I, S→M, which is unlike LO.

Conflict Detection: LP's conflict detection operates the same as EP's: using coherence messages to look for conflicts between transactions.

Validation: Like in EP, pessimistic conflict detection ensures that at any point, a running transaction has no conflicts with any other running transaction, so validation is a no-op.

Commit: Commit needs no special treatment: simply clear W **138** and R **132** bits and the store buffer, like in LO.

Abort: Rollback is also like that of LO: simply invalidate the write-set using the store buffer and clear the W and R bits and the store buffer.

Eager-Optimistic (EO)

The LP has the characteristics of: Like LO, aborts are very fast. Like EP, the use of pessimistic conflict detection reduces the number of "doomed" transactions. Like EP, some serializable schedules are not allowed and conflict detection must be performed on each cache miss.

The final combination of versioning and conflict detection is Eager-Optimistic (EO). EO may be a less than optimal choice for HTM systems: since new transactional versions are written in-place, other transactions have no choice but to notice conflicts as they occur (i.e., as cache misses occur). But since EO waits until commit time to detect conflicts, those transactions become "zombies," continuing to execute, wasting resources, yet are "doomed" to abort.

EO has proven to be useful in STMs and is implemented by Bartok-STM and McRT. A lazy versioning STM needs to

check its write buffer on each read to ensure that it is reading the most recent value. Since the write buffer is not a hardware structure, this is expensive, hence the preference for write-in-place eager versioning. Additionally, since checking for conflicts is also expensive in an STM, optimistic conflict detection offers the advantage of performing this operation in bulk.

Contention Management

How a transaction rolls back once the system has decided to abort that transaction has been described above, but, since a conflict involves two transactions, the topics of which transaction should abort, how that abort should be initiated, and when should the aborted transaction be retried need to be explored. These are topics that are addressed by Contention Management (CM), a key component of transactional memory. Described below are policies regarding how the systems initiate aborts and the various established methods of managing which transactions should abort in a conflict.

Contention Management Policies

A Contention Management (CM) Policy is a mechanism that determines which transaction involved in a conflict should abort and when the aborted transaction should be retried. For example, it is often the case that retrying an aborted transaction immediately does not lead to the best performance. Conversely, employing a back-off mechanism, which delays the retrying of an aborted transaction, can yield better performance. STMs first grappled with finding the best contention management policies and many of the policies outlined below were originally developed for STMs.

CM Policies draw on a number of measures to make decisions, including ages of the transactions, size of read- and write-sets, the number of previous aborts, etc. The combinations of measures to make such decisions are endless, but certain combinations are described below, roughly in order of increasing complexity.

To establish some nomenclature, first note that in a conflict there are two sides: the attacker and the defender. The attacker is the transaction requesting access to a shared memory location. In pessimistic conflict detection, the attacker is the transaction issuing the load or load exclusive. In optimistic, the attacker is the transaction attempting to validate. The defender in both cases is the transaction receiving the attacker's request.

An Aggressive CM Policy immediately and always retries either the attacker or the defender. In LO, Aggressive means that the attacker always wins, and so Aggressive is sometimes called committer wins. Such a policy was used for the earliest LO systems. In the case of EP, Aggressive can be either defender wins or attacker wins.

Restarting a conflicting transaction that will immediately experience another conflict is bound to waste work—namely interconnect bandwidth refilling cache misses. A Polite CM Policy employs exponential back off (but linear could also be used) before restarting conflicts. To prevent starvation, a situation where a process does not have resources allocated to it by the scheduler, the exponential back off greatly increases the odds of transaction success after some n retries.

Another approach to conflict resolution is to randomly abort the attacker or defender (a policy called Randomized). Such a policy may be combined with a randomized back off scheme to avoid unneeded contention.

However, making random choices, when selecting a transaction to abort, can result in aborting transactions that have completed "a lot of work", which can waste resources. To avoid such waste, the amount of work completed on the transaction can be taken into account when determining which transaction to abort. One measure of work could be a

transaction's age. Other methods include Oldest, Bulk TM, Size Matters, Karma, and Polka. Oldest is a simple time-stamp method that aborts the younger transaction in a conflict. Bulk TM uses this scheme. Size Matters is like Oldest but instead of transaction age, the number of read/ 5 written words is used as the priority, reverting to Oldest after a fixed number of aborts. Karma is similar, using the size of the write-set as priority. Rollback then proceeds after backing off a fixed amount of time. Aborted transactions keep their priorities after being aborted (hence the name Karma). 10 Polka works like Karma but instead of backing off a pre-defined amount of time, it backs off exponentially more each time.

Since aborting wastes work, it is logical to argue that stalling an attacker until the defender has finished their transaction would lead to better performance. Unfortunately, such a simple scheme easily leads to deadlock. 15

Deadlock avoidance techniques can be used to solve this problem. Greedy uses two rules to avoid deadlock. The first rule is, if a first transaction, T1, has lower priority than a second transaction, T0, or if T1 is waiting for another transaction, then T1 aborts when conflicting with T0. The second rule is, if T1 has higher priority than T0 and is not waiting, then T0 waits until T1 commits, aborts, or starts waiting (in which case the first rule is applied). Greedy 20 provides some guarantees about time bounds for executing a set of transactions. One EP design (LogTM) uses a CM policy similar to Greedy to achieve stalling with conservative deadlock avoidance.

Example MESI coherency rules provide for four possible states in which a cache line of a multiprocessor cache system may reside, M, E, S, and I, defined as follows: 25

Modified (M): The cache line is present only in the current cache, and is dirty; it has been modified from the value in main memory. The cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state. The write-back changes the line to the Exclusive state. 30

Exclusive (E): The cache line is present only in the current cache, but is clean; it matches main memory. It may be changed to the Shared state at any time, in response to a read request. Alternatively, it may be changed to the Modified state when writing to it. 35

Shared (S): Indicates that this cache line may be stored in other caches of the machine and is "clean"; it matches the main memory. The line may be discarded (changed to the Invalid state) at any time. 40

Invalid (I): Indicates that this cache line is invalid (unused).

TM coherency status indicators (R 132, W 138) may be provided for each cache line, in addition to, or encoded in the MESI coherency bits. An R 132 indicator indicates the current transaction has read from the data of the cache line, and a W 138 indicator indicates the current transaction has written to the data of the cache line. 45

In another aspect of TM design, a system is designed using transactional store buffers. U.S. Pat. No. 6,349,361 titled "Methods and Apparatus for Reordering and Renaming Memory References in a Multiprocessor Computer System," filed Mar. 31, 2000 and incorporated by reference, teaches a method for reordering and renaming memory references in a multiprocessor computer system having at least a first and a second processor. The first processor has a first private cache and a first buffer, and the second processor has a second private cache and a second buffer. The method includes the steps of, for each of a plurality of gated store requests received by the first processor to store 50

a datum, exclusively acquiring a cache line that contains the datum by the first private cache, and storing the datum in the first buffer. Upon the first buffer receiving a load request from the first processor to load a particular datum, the particular datum is provided to the first processor from among the data stored in the first buffer based on an in-order sequence of load and store operations. Upon the first cache receiving a load request from the second cache for a given datum, an error condition is indicated and a current state of at least one of the processors is reset to an earlier state when the load request for the given datum corresponds to the data stored in the first buffer. 5

The main implementation components of one such transactional memory facility are a transaction-backup register file for holding pre-transaction GR (general register) content, a cache directory to track the cache lines accessed during the transaction, a store cache to buffer stores until the transaction ends, and firmware routines to perform various complex functions. In this section a detailed implementation is described. 10

IBM zEnterprise EC12 Enterprise Server Embodiment

The IBM zEnterprise EC12 enterprise server introduces transactional execution (TX) in transactional memory, and is described in part in a paper, "Transactional Memory Architecture and Implementation for IBM System z" of Proceedings Pages 25-36 presented at MICRO-45, 1-5 Dec. 2012, Vancouver, British Columbia, Canada, available from IEEE Computer Society Conference Publishing Services (CPS), which is incorporated by reference. 15

Table 3 shows an example transaction. Transactions started with TBEGIN are not assured to ever successfully complete with TEND, since they can experience an aborting condition at every attempted execution, e.g., due to repeating conflicts with other CPUs. This requires that the program support a fallback path to perform the same operation non-transactionally, e.g., by using traditional locking schemes. This puts significant burden on the programming and software verification teams, especially where the fallback path is not automatically generated by a reliable compiler. 20

TABLE 3

Example Transaction Code			
loop	LHI	R0,0	*initialize retry count=0
	TBEGIN		*begin transaction
	JNZ	abort	*go to abort code if CC1=0
	LT	R1, lock	*load and test the fallback lock
	JNZ	lckbzy	*branch if lock busy
	... perform operation ...		
	TEND		*end transaction
		
lckbzy	TABORT		*abort if lock busy; this *resumes after TBEGIN
abort	JO	fallback	*no retry if CC=3
	AHI	R0, 1	*increment retry count
	CIJNL	R0,6, fallback	*give up after 6 attempts
	PPA	R0, TX	*random delay based on retry count
	... potentially wait for lock to become free ...		
	J	loop	*jump back to retry fallback
	OBTAIN	lock	*using Compare&Swap
	... perform operation ...		
	RELEASE	lock	
		

The requirement of providing a fallback path for aborted Transaction Execution (TX) transactions can be onerous. Many transactions operating on shared data structures are expected to be short, touch only a few distinct memory 65

locations, and use simple instructions only. For those transactions, the IBM zEnterprise EC12 introduces the concept of constrained transactions; under normal conditions, the CPU 114 assures that constrained transactions eventually end successfully, albeit without giving a strict limit on the number of necessary retries. A constrained transaction starts with a TBEGINC instruction and ends with a regular TEND. Implementing a task as a constrained or non-constrained transaction typically results in very comparable performance, but constrained transactions simplify software development by removing the need for a fallback path. IBM's Transactional Execution architecture is further described in *z/Architecture, Principles of Operation, Tenth Edition, SA22-7832-09* published Sep. 2012 from IBM, incorporated by reference.

A constrained transaction starts with the TBEGINC instruction. A transaction initiated with TBEGINC must follow a list of programming constraints; otherwise the program takes a non-filterable constraint-violation interruption. Exemplary constraints may include, but not be limited to: the transaction can execute a maximum of 32 instructions, all instruction text must be within 256 consecutive bytes of memory; the transaction contains only forward-pointing relative branches (i.e., no loops or subroutine calls); the transaction can access a maximum of 4 aligned octowords (an octoword is 32 bytes) of memory; and restriction of the instruction-set to exclude complex instructions like decimal or floating-point operations. The constraints are chosen such that many common operations like doubly linked list-insert/delete operations can be performed, including the very powerful concept of atomic compare-and-swap targeting up to 4 aligned octowords. At the same time, the constraints were chosen conservatively such that future CPU implementations can assure transaction success without needing to adjust the constraints, since that would otherwise lead to software incompatibility.

TBEGINC mostly behaves like XBEGIN in TSX or TBEGIN on IBM's zEC12 servers, except that the floating-point register (FPR) control and the program interruption filtering fields do not exist and the controls are considered to be zero. On a transaction abort, the instruction address is set back directly to the TBEGINC instead of to the instruction after, reflecting the immediate retry and absence of an abort path for constrained transactions.

Nested transactions are not allowed within constrained transactions, but if a TBEGINC occurs within a non-constrained transaction it is treated as opening a new non-constrained nesting level just like TBEGIN would. This can occur, e.g., if a non-constrained transaction calls a subroutine that uses a constrained transaction internally.

Since interruption filtering is implicitly off, all exceptions during a constrained transaction lead to an interruption into the operating system (OS). Eventual successful finishing of the transaction relies on the capability of the OS to page-in the at most 4 pages touched by any constrained transaction. The OS must also ensure time-slices long enough to allow the transaction to complete.

TABLE 4

Transaction Code Example	
TBEGINC	*begin constrained transaction
. . . perform operation . . .	
TEND	*end transaction

Table 4 shows the constrained-transactional implementation of the code in Table 3, assuming that the constrained transactions do not interact with other locking-based code. No lock testing is shown therefore, but could be added if constrained transactions and lock-based code were mixed.

When failure occurs repeatedly, software emulation is performed using Millicode as part of system firmware. Advantageously, constrained transactions have desirable properties because of the burden removed from programmers.

With reference to FIG. 3, the IBM zEnterprise EC12 processor introduced the transactional execution facility. The processor can decode 3 instructions per clock cycle; simple instructions are dispatched as single micro-ops, and more complex instructions are cracked into multiple micro-ops. The micro-ops (Uops 232b) are written into a unified issue queue 216, from where they can be issued out-of-order. Up to two fixed-point, one floating-point, two load/store, and two branch instructions can execute every cycle. A Global Completion Table (GCT) 232 holds every micro-op and a transaction nesting depth (TND) 232a. The GCT 232 is written in-order at decode time, tracks the execution status of each micro-op 232b, and completes instructions when all micro-ops 232b of the oldest instruction group have successfully executed.

The level 1 (L1) data cache 240 is a 96 KB (kilo-byte) 6-way associative cache with 256 byte cache-lines and 4 cycle use latency, coupled to a private 1 MB (mega-byte) 8-way associative 2nd-level (L2) data cache 268 with 7 cycles use-latency penalty for L1 240 misses. L1 240 cache is the cache closest to a processor and Ln cache is a cache at the nth level of caching. Both L1 240 and L2 268 caches are store-through. Six cores on each central processor (CP) chip share a 48 MB 3rd-level store-in cache, and six CP chips are connected to an off-chip 384 MB 4th-level cache, packaged together on a glass ceramic multi-chip module (MCM). Up to 4 multi-chip modules (MCMs) can be connected to a coherent symmetric multi-processor (SMP) system with up to 144 cores (not all cores are available to run customer workload).

Coherency is managed with a variant of the MESI protocol. Cache-lines can be owned read-only (shared) or exclusive; the L1 240 and L2 268 are store-through and thus do not contain dirty lines. The L3 272 and L4 caches (not shown) are store-in and track dirty states. Each cache is inclusive of all its connected lower level caches.

Coherency requests are called "cross interrogates" (XI) and are sent hierarchically from higher level to lower-level caches, and between the L4s. When one core misses the L1 240 and L2 268 and requests the cache line from its local L3 272, the L3 272 checks whether it owns the line, and if necessary sends an XI to the currently owning L2 268/L1 240 under that L3 272 to ensure coherency, before it returns the cache line to the requestor. If the request also misses the L3 272, the L3 272 sends a request to the L4 (not shown), which enforces coherency by sending XIs to all necessary L3s under that L4, and to the neighboring L4s. Then the L4 responds to the requesting L3 which forwards the response to the L2 268/L1 240.

Note that due to the inclusivity rule of the cache hierarchy, sometimes cache lines are XI'ed from lower-level caches due to evictions on higher-level caches caused by associativity overflows from requests to other cache lines. These XIs can be called "LRU XIs", where LRU stands for least recently used.

Making reference to yet another type of XI requests, Demote-XIs transition cache-ownership from exclusive into

read-only state, and Exclusive-XIs transition cache ownership from exclusive into invalid state. Demote-XIs and Exclusive-XIs need a response back to the XI sender. The target cache can “accept” the XI, or send a “reject” response if it first needs to evict dirty data before accepting the XI. The L1 **240**/L2 **268** caches are store through, but may reject demote-XIs and exclusive XIs if they have stores in their store queues that need to be sent to L3 before downgrading the exclusive state. A rejected XI will be repeated by the sender. Read-only-XIs are sent to caches that own the line read-only; no response is needed for such XIs since they cannot be rejected. The details of the SMP protocol are similar to those described for the IBM z10 by P. Mak, C. Walters, and G. Strait, in “IBM System z10 processor cache subsystem microarchitecture”, IBM Journal of Research and Development, Vol 53:1, 2009, which is incorporated by reference.

Transactional Instruction Execution

FIG. 3 depicts example components of an example CPU environment **112**, including a CPU **114** and caches/components with which it interacts (such as those depicted in FIGS. 1 and 2). The instruction decode unit **208** (IDU) keeps track of the current transaction nesting depth **212** (TND). When the IDU **208** receives a TBEGIN instruction, the nesting depth **212** is incremented, and conversely decremented on TEND instructions. The nesting depth **212** is written into the GCT **232** for every dispatched instruction. When a TBEGIN or TEND is decoded on a speculative path that later gets flushed, the IDU’s **208** nesting depth **212** is refreshed from the youngest GCT **232** entry that is not flushed. The transactional state is also written into the issue queue **216** for consumption by the execution units, mostly by the Load/Store Unit (LSU) **280**, which also has an effective address calculator **236** is included in the LSU **280**. The TBEGIN instruction may specify a transaction diagnostic block (TDB) for recording status information, should the transaction abort before reaching a TEND instruction.

Similar to the nesting depth, the IDU **208**/GCT **232** collaboratively track the access register/floating-point register (AR/FPR) modification masks through the transaction nest; the IDU **208** can place an abort request into the GCT **232** when an AR/FPR-modifying instruction is decoded and the modification mask blocks that. When the instruction becomes next-to-complete, completion is blocked and the transaction aborts. Other restricted instructions are handled similarly, including TBEGIN if decoded while in a constrained transaction, or exceeding the maximum nesting depth.

An outermost TBEGIN is cracked into multiple micro-ops depending on the GR-Save-Mask; each micro-op **232b** (including, for example uop **0**, uop **1**, and uop **2**) will be executed by one of the two fixed point units (FXUs) **220** to save a pair of GRs **228** into a special transaction-backup register file **224**, that is used to later restore the GR **228** content in case of a transaction abort. Also the TBEGIN spawns micro-ops **232b** to perform an accessibility test for the TDB if one is specified; the address is saved in a special purpose register for later usage in the abort case. At the decoding of an outermost TBEGIN, the instruction address and the instruction text of the TBEGIN are also saved in special purpose registers for a potential abort processing later on.

TEND and NTSTG are single micro-op **232b** instructions; NTSTG (non-transactional store) is handled like a normal store except that it is marked as non-transactional in the issue queue **216** so that the LSU **280** can treat it appropri-

ately. TEND is a no-op at execution time, the ending of the transaction is performed when TEND completes.

As mentioned, instructions that are within a transaction are marked as such in the issue queue **216**, but otherwise execute mostly unchanged; the LSU **280** performs isolation tracking as described in the next section.

Since decoding is in-order, and since the IDU **208** keeps track of the current transactional state and writes it into the issue queue **216** along with every instruction from the transaction, execution of TBEGIN, TEND, and instructions before, within, and after the transaction can be performed out-of order. It is even possible (though unlikely) that TEND is executed first, then the entire transaction, and lastly the TBEGIN executes. Program order is restored through the GCT **232** at completion time. The length of transactions is not limited by the size of the GCT **232**, since general purpose registers (GRs) **228** can be restored from the backup register file **224**.

During execution, the program event recording (PER) events are filtered based on the Event Suppression Control, and a PER TEND event is detected if enabled. Similarly, while in transactional mode, a pseudo-random generator may be causing the random aborts as enabled by the Transaction Diagnostics Control.

Tracking for Transactional Isolation

The Load/Store Unit **280** tracks cache lines that were accessed during transactional execution, and triggers an abort if an XI from another CPU (or an LRU-XI) conflicts with the footprint. If the conflicting XI is an exclusive or demote XI, the LSU **280** rejects the XI back to the L3 **272** in the hope of finishing the transaction before the L3 **272** repeats the XI. This “stiff-arming” is very efficient in highly contended transactions. In order to prevent hangs when two CPUs stiff-arm each other, a XI-reject counter is implemented, which triggers a transaction abort when a threshold is met.

The L1 cache directory **240** is traditionally implemented with static random access memories (SRAMs). For the transactional memory implementation, the valid bits **244** (64 rows×6 ways) of the directory have been moved into normal logic latches, and are supplemented with two more bits per cache line: the TX-read **248** and TX-dirty **252** bits.

The TX-read **248** bits are reset when a new outermost TBEGIN is decoded (which is interlocked against a prior still pending transaction). The TX-read **248** bit is set at execution time by every load instruction that is marked “transactional” in the issue queue. Note that this can lead to over-marking if speculative loads are executed, for example on a mispredicted branch path. The alternative of setting the TX-read **248** bit at load completion time was too expensive for silicon area, since multiple loads can complete at the same time, requiring many read-ports on the load-queue.

Stores execute the same way as in non-transactional mode, but a transaction mark is placed in the store queue (STQ) **260** entry of the store instruction. At write-back time, when the data from the STQ **260** is written into the L1 **240**, the TX-dirty bit **252** in the L1-directory **256** is set for the written cache line. Store write-back into the L1 **240** occurs only after the store instruction has completed, and at most one store is written back per cycle. Before completion and write-back, loads can access the data from the STQ **260** by means of store-forwarding; after write-back, the CPU **114** (FIG. 2) can access the speculatively updated data in the L1 **240**. If the transaction ends successfully, the TX-dirty bits **252** of all cache-lines are cleared, and also the TX-marks of not yet written stores are cleared in the STQ **260**, effectively turning the pending stores into normal stores.

On a transaction abort, all pending transactional stores are invalidated from the STQ **260**, even those already completed. All cache lines that were modified by the transaction in the L1 **240**, that is, have the TX-dirty bit **252** on, have their valid bits turned off, effectively removing them from the L1 **240** cache instantaneously.

The architecture requires that before completing a new instruction, the isolation of the transaction read- and write-set is maintained. This isolation is ensured by stalling instruction completion at appropriate times when XIs are pending; speculative out-of order execution is allowed, optimistically assuming that the pending XIs are to different addresses and not actually cause a transaction conflict. This design fits very naturally with the XI-vs-completion interlocks that are implemented on prior systems to ensure the strong memory ordering that the architecture requires.

When the L1 **240** receives an XI, L1 **240** accesses the directory to check validity of the XI'ed address in the L1 **240**, and if the TX-read bit **248** is active on the XI'ed line and the XI is not rejected, the LSU **280** triggers an abort. When a cache line with active TX-read bit **248** is LRU'ed from the L1 **240**, a special LRU-extension vector remembers for each of the 64 rows of the L1 **240** that a TX-read line existed on that row. Since no precise address tracking exists for the LRU extensions, any non-rejected XI that hits a valid extension row the LSU **280** triggers an abort. Providing the LRU-extension effectively increases the read footprint capability from the L1-size to the L2-size and associativity, provided no conflicts with other CPUs **114** (FIG. 1) against the non-precise LRU-extension tracking causes aborts.

The store footprint is limited by the store cache size (the store cache is discussed in more detail below) and thus implicitly by the L2 **268** size and associativity. No LRU-extension action needs to be performed when a TX-dirty **252** cache line is LRU'ed from the L1 **240**.

Store Cache

In prior systems, since the L1 **240** and L2 **268** are store-through caches, every store instruction causes an L3 **272** store access; with now 6 cores per L3 **272** and further improved performance of each core, the store rate for the L3 **272** (and to a lesser extent for the L2 **268**) becomes problematic for certain workloads. In order to avoid store queuing delays, a gathering store cache **264** had to be added, that combines stores to neighboring addresses before sending them to the L3 **272**.

For transactional memory performance, it is acceptable to invalidate every TX-dirty **252** cache line from the L1 **240** on transaction aborts, because the L2 **268** cache is very close (7 cycles L1 **240** miss penalty) to bring back the clean lines. However, it would be unacceptable for performance (and silicon area for tracking) to have transactional stores write the L2 **268** before the transaction ends and then invalidate all dirty L2 **268** cache lines on abort (or even worse on the shared L3 **272**).

The two problems of store bandwidth and transactional memory store handling can both be addressed with the gathering store cache **264**. The cache **232** is a circular queue of 64 entries, each entry holding 128 bytes of data with byte-precise valid bits. In non-transactional operation, when a store is received from the LSU **280**, the store cache checks whether an entry exists for the same address, and if so gathers the new store into the existing entry. If no entry exists, a new entry is written into the queue, and if the number of free entries falls under a threshold, the oldest entries are written back to the L2 **268** and L3 **272** caches.

When a new outermost transaction begins, all existing entries in the store cache are marked closed so that no new

stores can be gathered into them, and eviction of those entries to L2 **268** and L3 **272** is started. From that point on, the transactional stores coming out of the LSU **280** STQ **260** allocate new entries, or gather into existing transactional entries. The write-back of those stores into L2 **268** and L3 **272** is blocked, until the transaction ends successfully; at that point subsequent (post-transaction) stores can continue to gather into existing entries, until the next transaction closes those entries again.

The store cache is queried on every exclusive or demote XI, and causes an XI reject if the XI compares to any active entry. If the core is not completing further instructions while continuously rejecting XIs, the transaction is aborted at a certain threshold to avoid hangs.

The LSU **280** requests a transaction abort when the store cache overflows. The LSU **280** detects this condition when it tries to send a new store that cannot merge into an existing entry, and the entire store cache is filled with stores from the current transaction. The store cache is managed as a subset of the L2 **268**: while transactionally dirty lines can be evicted from the L1 **240**, they have to stay resident in the L2 **268** throughout the transaction. The maximum store footprint is thus limited to the store cache size of 64×128 bytes, and it is also limited by the associativity of the L2 **268**. Since the L2 **268** is 8-way associative and has 512 rows, it is typically large enough to not cause transaction aborts.

If a transaction aborts, the store cache is notified and all entries holding transactional data are invalidated. The store cache also has a mark per doubleword (8 bytes) whether the entry was written by a NTSTG instruction—those doublewords stay valid across transaction aborts.

Millicode-Implemented Functions

Traditionally, IBM mainframe server processors contain a layer of firmware called Millicode which performs complex functions like certain CISC instruction executions, interruption handling, system synchronization, and RAS. Millicode includes machine dependent instructions as well as instructions of the instruction set architecture (ISA) that are fetched and executed from memory similarly to instructions of application programs and the operating system (OS). Firmware resides in a restricted area of main memory that customer programs cannot access. When hardware detects a situation that needs to invoke Millicode, the instruction fetching unit **204** switches into "Millicode mode" and starts fetching at the appropriate location in the Millicode memory area. Millicode may be fetched and executed in the same way as instructions of the instruction set architecture (ISA), and may include ISA instructions.

For transactional memory, Millicode is involved in various complex situations. Every transaction abort invokes a dedicated Millicode sub-routine to perform the necessary abort steps. The transaction-abort Millicode starts by reading special-purpose registers (SPRs) holding the hardware internal abort reason, potential exception reasons, and the aborted instruction address, which Millicode then uses to store a TDB if one is specified. The TBEGIN instruction text is loaded from an SPR to obtain the GR-save-mask, which is needed for Millicode to know which GRs **238** to restore.

The CPU **114** (FIG. 2) supports a special Millicode-only instruction to read out the backup-GRs **224** and copy them into the main GRs **228**. The TBEGIN instruction address is also loaded from an SPR to set the new instruction address in the PSW (program status word) to continue execution after the TBEGIN once the Millicode abort sub-routine finishes. The PSW may later be saved as program-old PSW in case the abort is caused by a non-filtered program interruption.

The TABORT instruction may be Millicode implemented; when the IDU **208** decodes TABORT, it instructs the instruction fetch unit to branch into TABORT's Millicode, from which Millicode branches into the common abort sub-routine.

The Extract Transaction Nesting Depth (ETND) instruction may also be millicoded, since it is not performance critical; Millicode loads the current nesting depth out of a special hardware register and places it into a GR **228**. The PPA instruction is millicoded; it performs the optimal delay based on the current abort count provided by software as an operand to PPA, and also based on other hardware internal state.

For constrained transactions, Millicode may keep track of the number of aborts. The counter is reset to 0 on successful TEND completion, or if an interruption into the OS occurs (since it is not known if or when the OS will return to the program). Depending on the current abort count, Millicode can invoke certain mechanisms to improve the chance of success for the subsequent transaction retry. The mechanisms involve, for example, successively increasing random delays between retries, and reducing the amount of speculative execution to avoid encountering aborts caused by speculative accesses to data that the transaction is not actually using. As a last resort, Millicode can broadcast to other CPUs **114** (FIG. 2) to stop all conflicting work, retry the local transaction, before releasing the other CPUs **114** to continue normal processing. Multiple CPUs **114** must be coordinated to not cause deadlocks, so some serialization between Millicode instances on different CPUs **114** is required.

Various embodiments of the disclosure may be implemented in a data processing system suitable for storing and/or executing program code that includes at least one processor coupled directly or indirectly to memory elements through a system bus. The memory elements include, for instance, local memory employed during actual execution of the program code, bulk storage, and cache memory which provide temporary storage of at least some program code in order to reduce the number of times code must be retrieved from bulk storage during execution.

Input/Output or I/O devices (including, but not limited to, keyboards, displays, pointing devices, DASD (Direct Access Storage Device), tape, CDs (Compact Discs), DVDs (Digital Versatile Disc), thumb drives and other memory media, etc.) can be coupled to the system either directly or through intervening I/O controllers. Network adapters may also be coupled to the system to enable the data processing system to become coupled to other data processing systems or remote printers or storage devices through intervening private or public networks. Modems, cable modems, and Ethernet cards are just a few of the available types of network adapters.

A sequence of program instructions or a logical assembly of one or more interrelated modules defined by one or more computer readable program code means or logic direct the performance of one or more aspects of the present disclosure.

Although one or more examples have been provided herein, these are only examples. Many variations are possible without departing from the spirit of the present disclosure. For instance, processing environments other than the examples provided herein may include and/or benefit from one or more aspects of the present disclosure. Further, the environment need not be based on the z/Architecture®, but instead can be based on other architectures offered by, for instance, IBM®, Intel®, Sun Microsystems, as well as

others. Yet further, the environment can include multiple processors, be partitioned, and/or be coupled to other systems, as examples.

As used herein, the term "obtaining" includes, but is not limited to, fetching, receiving, having, providing, being provided, creating, developing, etc.

The capabilities of one or more aspects of the present disclosure can be implemented in software, firmware, hardware, or some combination thereof. At least one program storage device readable by a machine embodying at least one program of instructions executable by the machine to perform the capabilities of the present disclosure can be provided.

The flow diagrams depicted herein are just examples. There may be many variations to these diagrams or the steps (or operations) described therein without departing from the spirit of the disclosure. For instance, the steps may be performed in a differing order, or steps may be added, deleted, or modified. All of these variations are considered a part of the claimed disclosure.

Although preferred embodiments have been depicted and described in detail herein, it will be apparent to those skilled in the relevant art that various modifications, additions, substitutions and the like can be made without departing from the spirit of the disclosure, and these are, therefore, considered to be within the scope of the disclosure, as defined in the following claims.

Symmetric I/O Mode

Example multi processor (MP) systems include an Intel® MP system as described in, Intel® "Multiprocessor Specification", Version 1.4, May 1997, and Intel®, INT86 Instruction, "Wikipedia®, The Free Encyclopedia". Wikimedia Foundation, Inc. Aug. 3, 2013, both of which are hereby incorporated by reference.

Referring to FIG. 4, in the Intel® MP system as described in the Multiprocessor Specification paper (referenced above), some MP operating systems operate in Symmetric I/O Mode **300**. The Symmetric I/O Mode requires at least one I/O (input/output) APIC **304** (Advanced Programmable Interrupt Controller) to operate. In this mode, I/O interrupts are generated by the I/O APIC **304**. All interrupt lines are either masked or work together with the I/O APIC **304** in a mixed mode.

Again, referring to FIG. 4, an overview of Symmetric I/O Mode **300** according to Intel® MP Specification is depicted and described below. The APIC I/O unit **304** has general-purpose interrupt inputs **308** that can be individually programmed to different operating modes. The I/O APIC **304** interrupt line assignments are system implementation specific. The dotted line **307** depicts an interrupt path. The hardware must support a mode of operation in which the system can switch easily to Symmetric I/O mode from a PIC **309** (programmable interrupt controller) or a Virtual Wire mode. When the operating system is ready to switch to MP operation, it writes a 01H to the IMCR (interrupt mode configuration register) register, if that register is implemented, and enables I/O APIC Redirection Table entries. The hardware must not require any other action on the part of software to make the transition to Symmetric I/O mode. An example PIC **309** (shown in FIG. 4) can be the Intel® 8259 programmable interrupt controller designed for Intel® microprocessors.

Assignment of System Interrupts to the APIC Local Unit

The APIC local units **312** have two general-purpose interrupt inputs **314**, which are reserved for system interrupts. The APIC local units **312** are connected to CPUs **320**. The interrupt inputs **314** can be individually programmed to

different operating modes. Like the I/O APIC interrupt lines, the local APIC interrupt line assignments of a non-PC/AT compatible system, are system implementation specific.

Referring to FIG. 5, an example generic computer system 400 is shown for illustrative purposes. The computer system 400 of FIG. 5 has similar components to the computer system shown in FIGS. 1-3, and is directed to the embodiments of the disclosure described hereinbelow. The computer system 400 can accept an interrupt input to a plurality of processors, such as, processors (CPUs) 430-436. Types of interruptions can be processed on the CPU on which they occur.

The computer can include a generic I/O interface 456. The computer system 400 is connected to a data storage device 402 via an input/output interface(s) 456, which can include a program 404 embodied thereon. The program 404 can be representative of one or more programs communicating with the computer system 400 which can include, for example, instructions for executing read/write commands. In another example, the program 404 may be read from the data storage device 402 into the memory 410 using the I/O interface 456, for execution by one of the CPUs. Computer memory is generically represented by memory 410 which can include cache memory in the CPU's designated as TX CPU's, that is, the first, second, third, and fourth CPU 430, 432, 434, 436, respectively. The memory 410 can accept TX instructions 420. The computer system 400 can include network interfaces 454, and input/output (I/O) interface(s) 456. The I/O interface 456 can direct an interrupt 440 to the CPUs 430-436. The I/O interface 456 allows for input and output of data with an external device 458 that may be connected to the computing device. The network interface 454 may provide communications between the computing device and a computer network.

Referring to FIG. 5, in general, a first CPU 430 receives an interrupt request 440. The CPU can process a TX in a program context and a TX compatible routine in a supervisor context. In general, an interrupt can include a signal to a processor indicating an event needing immediate attention. An interrupt alerts the processor to a condition which can require an interruption of the program (current thread) which is being executed by the processor. Interruptions may be both hardware and software initiated. For instance, a controller can be initiated by either hardware interrupts or interrupt instructions in software. A software interrupt can be caused by an exceptional condition in the processor itself, or a special instruction in an instruction set which causes an interrupt when the special instruction is executed. A software program may directly cause an interruption by inadvertently (or intentionally) causing an exception to be recognized in an instruction, for example dividing by zero, overflowing a counter, or attempting to access protected memory. Additionally, software can cause an interruption that is intended to invoke the supervisory services of an operating system—commonly referred to as a supervisor call or system call. A supervisor-call interruption may be used by the program to cause the operating system to perform authorized functions that are not directly available to an application program, for example: performing I/O operations to hardware devices, allocating and/or freeing blocks of memory from a common system, e.g., “heap”, of memory, for example, or requesting timer services. Heap includes a computer memory management system which can dynamically allocate portions of memory to programs on request, and free memory for reuse when no longer needed.

Referring to FIG. 6, an embodiment 500 according to the disclosure includes a computer system (for example, as

depicted in FIGS. 1-4). The computer system can manage an interruption while a processor is executing a transaction in a transactional-execution (TX) mode. The computer system is a generic representation of a TX environment according to the embodiment of the disclosure.

Referring to FIG. 6, execution begins a transaction in a program context, which initiates 504 a processor into a TX mode, for example, a first processor 430 (FIG. 5) can be in a TX mode in a program context. A program context comprises state information about resources available to a program such as, registers, instruction counters (PC), or condition flags. A program's architected context includes hardware registers associated with an individual program. The OS of the computer system can dynamically dispatch a program on any CPU in the computing system, thus the context can be saved/restored by the OS. The context may include: the program-status word (including the instruction address (also known as, a program counter), condition code, or address space control); the general registers, access registers, floating-point registers, floating-point control register, other control registers, vector registers, CPU timer, clock comparator, and others.

The first processor 430 (which is in TX mode), can detect 512 an interruption request for an interruption 440 (FIG. 5). When the interruption is not associated with a TX compatible routine, as determined in block 512, the transaction is aborted 516, and changes are not committed 520 to supervisor resources. A TX compatible routine can be a class of interruptions that are compatible with transactional execution because the handling of the interruption causes a processor state to change that can be withdrawn if a transaction aborts.

In an example, an application program may request a supervisor service such as allocation of memory. To fulfill this request, the operating system may simply mete out a portion of its common heap space, and update some internal pointers in memory indicating that this memory is now assigned to an application. Currently, such a supervisor-call operation is prohibited in a TX implementation.

However, even though a supervisor operation is performed by the OS, it may only require the alteration of memory using conventional instructions (which can be abandoned if performed in a transaction that aborts). In such a case, the supervisor call operation can successfully complete in TX mode.

Other supervisor services such as performing an I/O operation may cause changes to the state of the processor and I/O devices that cannot be rescinded if the transaction fails. For example, the CPU cannot cause a record to be written to disk or printer, and then have it disappear if the TX that caused the record is subsequently aborted. Such supervisor operations use special privileged instructions, the attempted execution of which will still cause a TX to be aborted.

When the interruption is a TX compatible routine, as determined at block 512, the embodiment 500 accepts the interruption 440 without leaving TX mode, by the first processor 430, to execute 524 a TX compatible routine in a supervisor context (or supervisor mode) for the changing supervisor resources. A supervisor resource can include supervisor main storage locations.

In general, processor (CPUs) can support multiple modes of operation, for instance, a protected mode and a supervisor mode. The supervisor mode can be used by an operating system's kernel for low level tasks that need unrestricted

access to hardware. Protected mode can be used for tasks that have restricted or no access to supervisor state information.

The context switch according to the disclosure causes a CPU that is executing the transaction to interrupt to a supervisor routine in the same CPU. As a result, TX cache conflicts can continue to be tracked in the same CPU. Additionally, in TX mode and supervisor context (mode), memory addresses accessed during read-set and write-set tracking may be completed in both the TX and the supervisor modes.

A context switch can include a computer changing from one task (or process) to another. Context switching enables user-friendly multitasking. Thus, an application program that encounters a virtual address page fault, can context switch to a page-fault handler, for example, a supervisor routine for paging data from DASD to memory. For example, the program to supervisor context switches, saves context (state) of the application program at the point of an instruction being executed, and obtains context (state) of the page-fault handler to enable the page-fault handler to perform the paging function. After performing the paging operation, the page-fault handler can permit a context switch back to the application program, for example. The term "context" can refer to program state information such as data in registers and program counters at a specific time in a CPU.

The embodiment **500** executes **528** the TX compatible routine within the TX mode. Then, the processor returns **532** to the program context to complete the execution of the transaction.

Referring to FIG. 7, an embodiment **600** of the disclosure includes beginning execution **504** which initiates a processor into TX mode, of a transaction in a program context which was depicted in FIG. 6 in the embodiment **500**. However, the embodiment **600** proceeds to executing **604**, by a processor in TX mode, a supervisor call (SC, or also known as SVC) instruction for requesting a rescindable operation by a supervisor on a supervisor resource. In one example, the execution of the SC instruction can cause the interruption request. The SC instruction can be an instruction that requests manipulation of rescindable supervisor resources.

Referring to FIG. 7, in one embodiment a supervisor call (SC) instruction **610** can be an instruction (e.g., an Intel INT instruction (an assembly language instruction that generates a software interrupt)) wherein execution of the instruction includes providing a software interrupt number for causing the interrupt function pointed to by the SC instruction in an interrupt table to be executed.

In one embodiment, the processor can treat a supervisor main storage resource as transactional memory for the transaction. For example, supervisor resources can be used as transactional memory, for example saving program context in memory **1122** (FIG. 13).

Thereby, the embodiment(s) of the present disclosure allow transactional execution to survive SVC interruptions, when the program does not attempt to alter a processor/CPU state that cannot be restored. This allows certain operating-system functions to be invoked that would otherwise be restricted, including functions, such as, dynamic memory allocation, for example a call to malloc() (the parenthesis following the name in C designating a function) which is a C programming language feature of invoking the memory-allocation (e.g., malloc function). The allocation of memory from a common pool of memory (often known as a heap) can include a GETMAIN or malloc() command. The dynamic memory allocation may only update OS tables, for example. A printf() function in the C programming language

is called by a program to perform a formatted print operation to an output device such as a disk, printer, or terminal. For various types of devices, calls to the printf() function simply cause data to be buffered in memory until a block of a sufficient size has been collected. For each invocation of printf() that simply performs buffering, no I/O operation is performed. In one embodiment, the easily-restored state comprises transactional stores to memory, by the supervisor and temporary supervisor registers. In one embodiment, the transaction is always aborted if a control instruction such as an I/O instruction is executed, by the TX compatible routine, whose results cannot be withdrawn if the transaction is aborted.

Thereby, the present disclosure provides an embodiment(s) for accepting a transaction initiated interruption that changes a CPU state only if the TX completes (ends). An advantage is that the CPU can be restored to a pre-transactional state if the transaction is aborted. For example, in some cases, such as a supervisor-call (SVC) interruption, the altered state may only comprise information that can easily be restored (e.g., memory and general registers). Thus, in this example, in the embodiment(s) of the present disclosure, aborting a transaction simply because it executes a Supervisor Call instruction is unnecessary.

Referring to FIGS. 8-9, another embodiment **1000** according to the disclosure includes a computer system **1002** communicating with a network interface **1004**, and external devices **1008**, for example, a user terminal or a tape drive. The computer system also communicates with data storage **1012** including one or more programs **1014** embodied thereon. An I/O interface **1016** communicates with the above elements, as shown in FIG. 12.

Another memory storage device **1020** is in the computer system **1002** and communicates with the I/O interface **1016**. The memory storage device **1020** includes a shared cache memory **1028**, as well as one or more programs **1024**. Two CPUs **1032**, which each include local cache **1036**, communicate with the shared cache **1028** of the memory **1020**.

Turning to FIG. 9, an embodiment **1100** is depicted in flowchart which can be run as a program executing on the computer system **1002**. A program is executed **1104** causing a CPU to enter a transactional-execution (TX) mode by issuing **1108** a transaction-beginning instruction, such as a TBEGIN instruction.

As defined in the IBM z/Architecture Principles of Operation, the TBEGIN instruction sets a condition code to zero to indicate that it has successfully initiated transactional execution. If a transaction is aborted, the IBM z/Architecture implementation causes control to return to the instruction following the TBEGIN with the condition code set to a non-zero value. The instruction following the TBEGIN is expected to be a conditional-branching instruction **1112** that will transfer control to an abort-handler routine if the condition code is nonzero. Other architectures may designate a location of the abort handler by other means such as a part of the transaction-beginning instruction.

When the transaction has successfully been initiated, the conditional branch instruction **1112** will not detect an abort, thus control will pass to the following instructions **1116** which may include a SUPERVISOR CALL (SVC) instruction to invoke **1120** a supervisory routine. In other architectures and operating systems, such an instruction may be referred to as an INT instruction or a system-call instruction. As currently defined in the IBM z/Architecture Principles of Operation, execution of an SVC instruction in the TX mode causes the transaction to be aborted. However, the embodi-

ment **1100** includes an enhancement wherein the transaction is allowed to continue by having the CPU enter the supervisor state **1120**.

As a part of entering **1120** the supervisor state, the hardware implicitly saves **1124** the program's context (also known as its original context), calling program's program-status word, including the program's instruction address (also known as program counter), the condition code, address-space controls, addressing mode, problem/supervisor state indication and other controls. The hardware then passes control to the operating-system's (OS's) SVC interruption handler. The OS's SVC interruption handler code saves any architected register context of the calling program that might be changed by the supervisor routine **1124**.

A supervisor routine may provide services to the calling program such as the allocation of memory from a common pool of memory (often known as a heap) commonly known as GETMAIN or malloc(). A supervisor service may also cause an input/output operation to be performed to a disk, tape, printer, or network device. The former type of service may only update OS tables in memory, for example, indicating which portions of the heap are allocated to a process, and as such services may update memory locations, they are inherently rescindable if the transaction is aborted. The latter type of service may cause immutable changes to the computer which cannot be rescinded if the transaction aborts, for example, a record written to a disk cannot be un-written if the transaction is aborted. Certain types of privileged instructions may cause such immutable changes to the computer, for example, the START SUBCHANNEL instruction which initiates an I/O operation, or the LOAD CONTROL instruction which may alter a control register that enables I/O interruptions.

Continuing at FIG. 9, the supervisor routine **1140** performs its processing in the supervisor state **1128** unless a restricted instruction is executed **1132**, in which case the transaction is aborted, and control is passed to the abort-handler routine (which is the instruction following **1108**). Furthermore, the supervisor routine may encounter numerous other reasons to be aborted **1136** such as a storage conflict with another CPU, an interrupt such as a timer expiration, or other reasons in which case control is passed to the abort-handler routine (i.e., following **1108**). Assuming no aborting conditions **1132** or **1136** are discovered, the supervisor routine **1140** continues to completion, at which time, the calling programs context is restored **1144** and control is passed back to the calling program **1148** to resume the program in its original context (e.g., registers and PSW) **1152**.

The calling program **1156** continues in the TX mode **1158** unless a reason to abort is encountered **1160**. As in the supervisor state, such a reason may include the attempted execution of a restricted instruction, a conflict with storage accessed from another CPU, a timer expiration, an I/O interruption, and so forth. Transactional execution may also be aborted if a program-exception condition is recognized, such an exception condition may be an access exception to memory, a numeric overflow condition, a divide-by-zero condition, an operation exception, and so forth. If an abort condition is detected, control is passed to the abort-handler routine (i.e., following **1108**). Assuming no reason to abort is encountered, the transaction eventually completes execution **1164**, and all of the transactional stores made during TX are committed to memory **1168** such that they are now visible to other CPUs and the I/O subsystem.

Regardless of whether the CPU was in a problem state (also known as the application state) or supervisor state (also

known as system state), if the transaction was aborted, control is passed to the transaction-abort handler. In the z/Architecture implementation, the transaction-abort handler receives control at the instruction following the transaction-beginning instruction **1108** with the condition code set to a nonzero value. In other embodiments, the transaction-abort handler may be at another location designated by the transaction-beginning instruction. In the z/Architecture implementation, the condition code is checked for a nonzero value **1112** to determine if the transaction was aborted. If so, all transactional stores that occurred during transactional-execution are discarded **1172**, selected registers of the program's register context are restored, and the transaction-abort-handler code is invoked to attempt to recover from the aborting condition **1176**.

Referring to at least FIG. 4, the present invention may be a system, a method, and/or a computer program product **405**. The computer program product may include a computer readable storage medium **402** (or media) having computer readable program instructions **404** thereon for causing a processor to carry out aspects of the present invention.

The computer readable storage medium can be a tangible device that can retain and store instructions for use by an instruction execution device. The computer readable storage medium may be, for example, but is not limited to, an electronic storage device, a magnetic storage device, an optical storage device, an electromagnetic storage device, a semiconductor storage device, or any suitable combination of the foregoing. A non-exhaustive list of more specific examples of the computer readable storage medium includes the following: a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), a static random access memory (SRAM), a portable compact disc read-only memory (CD-ROM), a digital versatile disk (DVD), a memory stick, a floppy disk, a mechanically encoded device such as punch-cards or raised structures in a groove having instructions recorded thereon, and any suitable combination of the foregoing. A computer readable storage medium, as used herein, is not to be construed as being transitory signals per se, such as radio waves or other freely propagating electromagnetic waves, electromagnetic waves propagating through a waveguide or other transmission media (e.g., light pulses passing through a fiber-optic cable), or electrical signals transmitted through a wire.

Computer readable program instructions described herein can be downloaded to respective computing/processing devices from a computer readable storage medium or to an external computer or external storage device via a network, for example, the Internet, a local area network, a wide area network and/or a wireless network. The network may comprise copper transmission cables, optical transmission fibers, wireless transmission, routers, firewalls, switches, gateway computers and/or edge servers. A network adapter card or network interface in each computing/processing device receives computer readable program instructions from the network and forwards the computer readable program instructions for storage in a computer readable storage medium within the respective computing/processing device.

Computer readable program instructions for carrying out operations of the present invention may be assembler instructions, instruction-set-architecture (ISA) instructions, machine instructions, machine dependent instructions, microcode, firmware instructions, state-setting data, or either source code or object code written in any combination of one or more programming languages, including an object

oriented programming language such as Smalltalk, C++ or the like, and conventional procedural programming languages, such as the "C" programming language or similar programming languages. The computer readable program instructions may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider). In some embodiments, electronic circuitry including, for example, programmable logic circuitry, field-programmable gate arrays (FPGA), or programmable logic arrays (PLA) may execute the computer readable program instructions by utilizing state information of the computer readable program instructions to personalize the electronic circuitry, in order to perform aspects of the present invention.

Aspects of the present invention are described herein with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems), and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer readable program instructions.

These computer readable program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks. These computer readable program instructions may also be stored in a computer readable storage medium that can direct a computer, a programmable data processing apparatus, and/or other devices to function in a particular manner, such that the computer readable storage medium having instructions stored therein comprises an article of manufacture including instructions which implement aspects of the function/act specified in the flowchart and/or block diagram block or blocks.

The computer readable program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other device to cause a series of operational steps to be performed on the computer, other programmable apparatus or other device to produce a computer implemented process, such that the instructions which execute on the computer, other programmable apparatus, or other device implement the functions/acts specified in the flowchart and/or block diagram block or blocks.

The flowchart and block diagrams in the Figures illustrate the architecture, functionality, and operation of possible implementations of systems, methods, and computer program products according to various embodiments of the present invention. In this regard, each block in the flowchart or block diagrams may represent a module, segment, or portion of instructions, which comprises one or more

executable instructions for implementing the specified logical function(s). In some alternative implementations, the functions noted in the block may occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and combinations of blocks in the block diagrams and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts or carry out combinations of special purpose hardware and computer instructions.

While embodiments of the present disclosure has been particularly shown and described with respect to preferred embodiments thereof, it will be understood by those skilled in the art that changes in forms and details may be made without departing from the spirit and scope of the present application. It is therefore intended that the present disclosure not be limited to the exact forms and details described and illustrated herein, but falls within the scope of the appended claims.

What is claimed is:

1. A computer implemented method for managing an interruption while a processor is executing a transaction in a transactional-execution (TX) mode, the method comprising:

executing, by a processor initiated into a TX mode by the executing, a transaction in a program context;

detecting, by the processor in the TX mode, an interruption request for an interruption;

based on the interruption being a TX compatible routine, accepting the interruption by the processor to execute the TX compatible routine in a supervisor context for changing supervisor resources;

executing the TX compatible routine within the TX mode;

executing, by the processor in the TX mode, a supervisor call (SC) instruction for requesting a rescindable operation by a supervisor on a supervisor resource, execution of the SC instruction causing the interruption request; and

returning to the program context to complete execution of the transaction.

2. The method of claim 1, wherein the supervisor resource comprises main storage.

3. The method of claim 2, wherein the processor uses a supervisor main storage resource as transactional memory for the transaction.

4. The method of claim 1, wherein the SC instruction is an instruction that requests manipulation of rescindable supervisor resources.

5. The method of claim 1, wherein the SC instruction is an assembly language instruction that generates a software interrupt, wherein execution of the SC instruction comprises providing a software interrupt number for causing an interrupt function pointed to by the SC instruction in an interrupt table to be executed.

6. The method of claim 1, further comprising:

based on the interruption being a non TX compatible routine, aborting the transaction and not-committing changes to the supervisor resources.

* * * * *